```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""\
v0_7_convergence_families_v23.py

Purpose
-------
Evidence step v19: add a physically meaningful, basis-invariant
perturbation robustness test.

This version keeps the fully reproducible, statistical–numerical protocol
(no model training)
and the REAL vs spectrum-matched Haar NULL baseline, while upgrading the
perturbation analysis
to use a 2D-subspace overlap metric (projector overlap) for near-
degenerate eigenpairs.

Key features
------------
- REAL vs NULL baseline: spectrum-matched Haar eigenbasis for NULL.
- Fixed-fraction stable selection per model (v15).
- Fine + coarse signatures (global quantiles).
- Perturbation robustness (v21.1) + Open-system (v23): apply the *same*
small Hermitian perturbation ΔH in the
  computational basis to both H_REAL and H_NULL, re-diagonalize, then
track each baseline
  near-degenerate 2D manifold to the best-matching *neighbor* pair in the
perturbed spectrum.
  Robustness is quantified by:

    (i)   pair_retention_rate: whether the best-matching neighbor gap
remains < eps
    (ii)  subspace_overlap: 0.5 * Tr(P P') = 0.5 * ||U^† U'||_F^2,
invariant to basis
    (iii) optional coarse-signature retention and feature drifts
(secondary)

Key refinements vs v17
----------------------
1) Perturbation robustness is now evidence-grade: basis-invariant 2D
subspace overlap,
   instead of relying on entropy drift under re-sampled Haar bases.
2) NULL perturbation is performed by constructing H_NULL = U diag(E) U^†
(Haar U) and
   applying the same ΔH in computational basis before re-diagonalizing.
3) Reporting includes overlap quantiles for interpretability.

Dependencies
------------
Only numpy (no Qiskit, no SciPy).

Notes on scaling
----------------
- For n_qubits=4, entropy ranges up to log2(d)=4 bits, so absolute
medians shift.
```

```
  Interpretations rely on REAL-NULL deltas/effect sizes and batch
convergence, not raw levels.
"""

from __future__ import annotations

import argparse
import math
import time
import hashlib
from dataclasses import dataclass
from typing import List, Tuple, Iterable, Dict, Optional, Any
from collections import Counter
from itertools import product

import numpy as np


# ----------------------------
# Utilities
# ----------------------------

def rng_from_seed(seed: int) -> np.random.Generator:
    return np.random.default_rng(int(seed))


def stable_hash_int(s: str) -> int:
    """Deterministic int hash independent of Python's hash
randomization."""
    h = hashlib.sha256(s.encode("utf-8")).hexdigest()
    return int(h[:16], 16)


def bin_index(x: float, step: float) -> int:
    if step <= 0:
        raise ValueError("step must be > 0")
    return int(math.floor(float(x) / float(step) + 1e-12))


def clamp_int(x: int, lo: int, hi: int) -> int:
    return max(lo, min(hi, x))


def jaccard(a: Iterable, b: Iterable) -> float:
    sa, sb = set(a), set(b)
    if not sa and not sb:
        return 1.0
    return len(sa & sb) / max(1, len(sa | sb))


def make_quantile_edges(values: np.ndarray, q_bins: int) -> np.ndarray:
    """
    Quantile edges for q_bins categories.
    Returns length q_bins+1, with edges[0]=-inf and edges[-1]=+inf.
    """
    if q_bins < 2:
```

```
        raise ValueError("q_bins must be >= 2")
    v = np.asarray(values, dtype=float)
    if v.size == 0:
        edges = np.linspace(0.0, 1.0, q_bins + 1)
    else:
        qs = np.linspace(0.0, 1.0, q_bins + 1)
        try:
            edges = np.quantile(v, qs, method="linear")
        except TypeError:
            edges = np.quantile(v, qs)
    edges = np.asarray(edges, dtype=float)
    edges[0] = -np.inf
    edges[-1] = np.inf
    edges = np.maximum.accumulate(edges)
    return edges


def quantile_bin(x: float, edges: np.ndarray, q_bins: int) -> int:
    idx = int(np.searchsorted(edges, float(x), side="right") - 1)
    return clamp_int(idx, 0, q_bins - 1)


def safe_median(x: np.ndarray) -> float:
    x = np.asarray(x, dtype=float)
    if x.size == 0:
        return 0.0
    return float(np.median(x))


# ----------------------------
# Exact binomial tail (no SciPy)
# ----------------------------

def _log_choose(n: int, k: int) -> float:
    return math.lgamma(n + 1) - math.lgamma(k + 1) - math.lgamma(n - k +
1)


def _log_binom_pmf(k: int, n: int, p: float) -> float:
    if k < 0 or k > n:
        return -math.inf
    p = float(p)
    if p <= 0.0:
        return 0.0 if k == 0 else -math.inf
    if p >= 1.0:
        return 0.0 if k == n else -math.inf
    return _log_choose(n, k) + k * math.log(p) + (n - k) * math.log(1.0 -
p)


def _logsumexp(log_terms: List[float]) -> float:
    m = max(log_terms)
    if m == -math.inf:
        return -math.inf
    s = sum(math.exp(t - m) for t in log_terms)
    return m + math.log(s)
```

```python
def binom_tail_ge(k: int, n: int, p: float) -> float:
    """Exact tail probability P(X >= k), X ~ Binomial(n, p)."""
    if k <= 0:
        return 1.0
    if k > n:
        return 0.0
    log_terms = [_log_binom_pmf(x, n, p) for x in range(k, n + 1)]
    return float(math.exp(_logsumexp(log_terms)))


# ---------------------------
# Pauli operator cache (dimension dependent)
# ---------------------------

PAULIS = ["I", "X", "Y", "Z"]
PAULI_MATS = {
    "I": np.array([[1, 0], [0, 1]], dtype=complex),
    "X": np.array([[0, 1], [1, 0]], dtype=complex),
    "Y": np.array([[0, -1j], [1j, 0]], dtype=complex),
    "Z": np.array([[1, 0], [0, -1]], dtype=complex),
}

def kron_n(mats: List[np.ndarray]) -> np.ndarray:
    out = mats[0]
    for m in mats[1:]:
        out = np.kron(out, m)
    return out


def precompute_pauli_ops(n_qubits: int) -> Tuple[List[str],
List[np.ndarray]]:
    """
    Precompute all n-qubit Pauli operators (excluding all-I) in a fixed
order.
    Returned labels are strings like "IXYZ". Operators are dense (d x d)
complex arrays.
    """
    labels: List[str] = []
    ops: List[np.ndarray] = []
    for tup in product(PAULIS, repeat=n_qubits):
        if all(p == "I" for p in tup):
            continue
        lab = "".join(tup)
        labels.append(lab)
        ops.append(kron_n([PAULI_MATS[p] for p in tup]))
    return labels, ops


@dataclass(frozen=True)
class PauliCache:
    n_qubits: int
    d: int
    labels: List[str]
    ops: List[np.ndarray]
```

```
    @staticmethod
    def build(n_qubits: int) -> "PauliCache":
        d = 2 ** int(n_qubits)
        labels, ops = precompute_pauli_ops(int(n_qubits))
        return PauliCache(n_qubits=int(n_qubits), d=d, labels=labels,
ops=ops)


def build_random_pauli_hamiltonian_cached(cache: PauliCache, n_terms:
int, seed: int) -> np.ndarray:
    rng = rng_from_seed(seed)
    d = cache.d
    H = np.zeros((d, d), dtype=complex)

    m = len(cache.ops)
    # Sample with replacement; coefficients uniform in [-1, 1]
    idxs = rng.integers(0, m, size=int(n_terms))
    coeffs = rng.uniform(-1.0, 1.0, size=int(n_terms)).astype(float)
    for k in range(int(n_terms)):
        H = H + float(coeffs[k]) * cache.ops[int(idxs[k])]

    H = 0.5 * (H + H.conj().T)
    return H


def haar_random_unitary(d: int, rng: np.random.Generator) -> np.ndarray:
    Z = (rng.normal(size=(d, d)) + 1j * rng.normal(size=(d, d))) /
np.sqrt(2.0)
    Q, R = np.linalg.qr(Z)
    diag = np.diag(R)
    ph = diag / np.abs(diag)
    Q = Q * ph
    return Q


def null_haar_basis_hamiltonian(evals: np.ndarray, seed: int) ->
np.ndarray:
    rng = rng_from_seed(stable_hash_int(f"NULL_HAAR_BASIS|{seed}"))
    d = evals.size
    U = haar_random_unitary(d, rng)
    H = U @ np.diag(evals) @ U.conj().T
    H = 0.5 * (H + H.conj().T)
    return H



def null_haar_basis_eigs(evals: np.ndarray, seed: int, tag: str =
"NULL_HAAR_BASIS") -> Tuple[np.ndarray, np.ndarray]:
    """Return (evals, evecs) for the Haar-basis NULL without re-
diagonalizing a full matrix."""
    rng = rng_from_seed(stable_hash_int(f"{tag}|{seed}"))
    d = evals.size
    U = haar_random_unitary(d, rng)
    return np.array(evals, dtype=float), U
```

```
def amplitude_entropy_bits(state: np.ndarray, eps: float = 1e-12) ->
float:
    p = np.abs(state) ** 2
    p = p / (p.sum() + eps)
    p = np.clip(p, eps, 1.0)
    return float(-np.sum(p * np.log2(p)))


def dominant_mask_by_mass(p: np.ndarray, keep_mass: float) -> np.ndarray:
    p = np.asarray(p, dtype=float)
    keep_mass = float(keep_mass)
    if p.size == 0:
        return np.zeros_like(p, dtype=bool)
    keep_mass = max(0.0, min(1.0, keep_mass))
    order = np.argsort(-p, kind="mergesort")
    cum = 0.0
    mask = np.zeros(p.size, dtype=bool)
    for idx in order:
        mask[idx] = True
        cum += float(p[idx])
        if cum >= keep_mass and np.any(mask):
            break
    if not np.any(mask):
        mask[order[0]] = True
    return mask


def leakage_proxy_from_state(state: np.ndarray, keep_mask: np.ndarray,
eps: float = 1e-12) -> float:
    p = np.abs(state) ** 2
    p = p / (p.sum() + eps)
    kept = float(np.sum(p[keep_mask])) if np.any(keep_mask) else 0.0
    return float(max(0.0, 1.0 - kept))


def leakage_proxy_fast(
    evals: np.ndarray,
    evecs: np.ndarray,
    i: int,
    j: int,
    times: List[float],
    keep_mass: float,
) -> Tuple[float, int]:
    vi = evecs[:, i]
    vj = evecs[:, j]
    psi0 = (vi + vj) / np.sqrt(2.0)

    p0 = np.abs(psi0) ** 2
    p0 = p0 / max(1e-12, float(np.sum(p0)))
    keep_mask = dominant_mask_by_mass(p0, keep_mass=keep_mass)
    dom = int(np.sum(keep_mask))
    dom = max(1, dom)

    Ei = float(evals[i])
    Ej = float(evals[j])
```

```python
    leaks = []
    for t in times:
        ph_i = np.exp(-1j * Ei * float(t))
        ph_j = np.exp(-1j * Ej * float(t))
        psi_t = (ph_i * vi + ph_j * vj) / np.sqrt(2.0)
        leaks.append(leakage_proxy_from_state(psi_t, keep_mask))
    return float(np.mean(leaks)), dom


# ----------------------------
# Signatures / Candidates
# ----------------------------

SigAbs = Tuple[int, int, int, int]             # (dom, ent_bin_i,
ent_bin_j, leak_bin)
SigQGlobal = Tuple[int, int, int, int]         # (dom, ent_q_lo,
ent_q_hi, leak_bin)
SigQGCoarse = Tuple[int, int, int, int]        # (dom_bin, ent_q_lo,
ent_q_hi, leak_bin_coarse)


def signature_key_abs(dom: int, ent_i: float, ent_j: float, leak: float,
ent_step: float, leak_step: float) -> SigAbs:
    bi = bin_index(ent_i, ent_step)
    bj = bin_index(ent_j, ent_step)
    lo, hi = (bi, bj) if bi <= bj else (bj, bi)
    lb = bin_index(leak, leak_step)
    return (int(dom), int(lo), int(hi), int(lb))


def signature_key_q_global(dom: int, ent_i: float, ent_j: float, leak:
float, edges: np.ndarray, q_bins: int, leak_step: float) -> SigQGlobal:
    qi = quantile_bin(ent_i, edges, q_bins)
    qj = quantile_bin(ent_j, edges, q_bins)
    lo, hi = (qi, qj) if qi <= qj else (qj, qi)
    lb = bin_index(leak, leak_step)
    return (int(dom), int(lo), int(hi), int(lb))


def dom_to_bin_ratio(dom: int, d: int) -> int:
    """
    Dimension-aware coarse binning for dom_count using dom/d ratio.
    Bins: [0..0.25], (0.25..0.5], (0.5..0.75], (0.75..1.0]
    Returns an integer in {0,1,2,3}.
    """
    dom = int(max(1, dom))
    d = int(max(1, d))
    r = float(dom) / float(d)
    if r <= 0.25:
        return 0
    if r <= 0.50:
        return 1
    if r <= 0.75:
        return 2
    return 3
```

```python
def leak_bin_coarse_from_fine(leak_bin: int) -> int:
    # Collapse fine leakage bins: 0 -> 0, 1 -> 1, >=2 -> 2
    lb = int(leak_bin)
    if lb <= 0:
        return 0
    if lb == 1:
        return 1
    return 2


def signature_key_qg_coarse(
    dom: int,
    ent_i: float,
    ent_j: float,
    leak: float,
    d: int,
    edges: np.ndarray,
    q_bins_coarse: int,
    leak_step_fine: float
) -> SigQGCoarse:
    qi = quantile_bin(ent_i, edges, q_bins_coarse)
    qj = quantile_bin(ent_j, edges, q_bins_coarse)
    lo, hi = (qi, qj) if qi <= qj else (qj, qi)

    lb_fine = bin_index(leak, leak_step_fine)
    lb = leak_bin_coarse_from_fine(lb_fine)
    return (int(dom_to_bin_ratio(dom, d)), int(lo), int(hi), int(lb))


@dataclass(frozen=True)
class Candidate:
    seed: int
    model: str
    i: int
    j: int
    delta_e: float
    gap_out: float
    iso_log: float
    ent_i: float
    ent_j: float
    dom: int
    leak: float
    score: float
    sig_abs: SigAbs
    sig_qg: SigQGlobal
    sig_qg_coarse: SigQGCoarse


def interestingness_score(dom: int, leak: float, dom_weight: float = 0.6,
leak_weight: float = 0.4) -> float:
    dom = max(1, int(dom))
    dom_term = 1.0 / (1.0 + float(dom))
    leak_term = 1.0 - float(leak)
    s = dom_weight * dom_term + leak_weight * leak_term
```

```
    return float(max(0.0, min(1.0, s)))


def find_neighbor_pairs(evals: np.ndarray, eps: float) -> List[Tuple[int,
int, float]]:
    pairs = []
    for k in range(len(evals) - 1):
        de = float(abs(evals[k + 1] - evals[k]))
        if de < float(eps):
            pairs.append((k, k + 1, de))
    return pairs


def generate_candidates_for_seed(
    model: str,
    seed: int,
    cache: PauliCache,
    n_terms: int,
    eps_neighbor: float,
    ent_step: float,
    leak_step: float,
    times: List[float],
    keep_mass: float,
    iso_eps: float,
) -> List[Candidate]:
    """
    Generate candidates for one seed under REAL or NULL_HAAR_BASIS
(spectrum-matched).
    Quantile signatures are assigned later at the batch level (pooled
edges).
    """
    H_real = build_random_pauli_hamiltonian_cached(cache, n_terms, seed)

    if model == "REAL":
        H = H_real
        evals, evecs = np.linalg.eigh(H)
    elif model == "NULL_HAAR_BASIS":
        evals_real, _ = np.linalg.eigh(H_real)
        H = null_haar_basis_hamiltonian(evals_real, seed=seed)
        evals, evecs = np.linalg.eigh(H)
    else:
        raise ValueError(f"Unknown model: {model}")

    pairs = find_neighbor_pairs(evals, eps_neighbor)
    if not pairs:
        return []

    cands: List[Candidate] = []
    for (i, j, de) in pairs:
        # Gap/isolation metrics for neighbor pair (i,i+1)
        left_gap = float(abs(evals[i] - evals[i-1])) if i > 0 else
float('inf')
        right_gap = float(abs(evals[j+1] - evals[j])) if (j + 1) <
len(evals) else float('inf')
        gap_out = min(left_gap, right_gap)
        if not np.isfinite(gap_out):
```

```
            gap_out = left_gap if np.isfinite(left_gap) else (right_gap
if np.isfinite(right_gap) else 0.0)
        denom = max(float(de), float(iso_eps))
        iso_log = float(math.log10(gap_out / denom)) if (gap_out > 0.0
and denom > 0.0) else float('nan')
        ent_i = amplitude_entropy_bits(evecs[:, i])
        ent_j = amplitude_entropy_bits(evecs[:, j])
        leak, dom = leakage_proxy_fast(evals, evecs, i, j, times,
keep_mass=keep_mass)
        score = interestingness_score(dom, leak)
        sig_abs = signature_key_abs(dom, ent_i, ent_j, leak, ent_step,
leak_step)
        cands.append(
            Candidate(
                seed=seed,
                model=model,
                i=i, j=j, delta_e=de,
                gap_out=gap_out, iso_log=iso_log,
                ent_i=ent_i, ent_j=ent_j,
                dom=dom, leak=leak, score=score,
                sig_abs=sig_abs,
                sig_qg=(0, 0, 0, 0),           # filled later
                sig_qg_coarse=(0, 0, 0, 0),    # filled later
            )
        )
    return cands


def assign_quantile_signatures(
    cands: List[Candidate],
    d: int,
    edges_fine: np.ndarray,
    q_bins_fine: int,
    edges_coarse: np.ndarray,
    q_bins_coarse: int,
    leak_step: float
) -> List[Candidate]:
    out: List[Candidate] = []
    for c in cands:
        sig_qg = signature_key_q_global(c.dom, c.ent_i, c.ent_j, c.leak,
edges_fine, q_bins_fine, leak_step)
        sig_qg_coarse = signature_key_qg_coarse(c.dom, c.ent_i, c.ent_j,
c.leak, d, edges_coarse, q_bins_coarse, leak_step)
        out.append(Candidate(**{**c.__dict__, "sig_qg": sig_qg,
"sig_qg_coarse": sig_qg_coarse}))
    return out


# ----------------------------
# Enrichment + batch summaries
# ----------------------------

@dataclass(frozen=True)
class FamRow:
    sig: Tuple[int, int, int, int]
    overall: int
```

```
    stable: int
    expected: float
    p_tail: float
    neglog10_p: float
    enrichment: float

# ----------------------------
# Perturbation robustness (v21.1)
# ----------------------------

def _subspace_overlap_2d(U: np.ndarray, Up: np.ndarray) -> float:
    """Basis-invariant overlap between 2D subspaces span(U) and
span(Up)."""
    # overlap = 0.5 * Tr(P P') = 0.5 * ||U^† U'||_F^2 for orthonormal
columns.
    M = U.conj().T @ Up
    return float(0.5 * np.sum(np.abs(M) ** 2))


def _best_match_neighbor_pair(evals_p: np.ndarray, Ei: float, Ej: float)
-> Tuple[int, int, float]:
    """Pick k,k+1 in perturbed spectrum that best matches (Ei,Ej) (allow
swap)."""
    d = int(evals_p.size)
    if d < 2:
        return 0, 0, float("inf")
    best_k = 0
    best_cost = float("inf")
    for k in range(d - 1):
        a = float(evals_p[k])
        b = float(evals_p[k + 1])
        c1 = (a - Ei) ** 2 + (b - Ej) ** 2
        c2 = (a - Ej) ** 2 + (b - Ei) ** 2
        c = c1 if c1 <= c2 else c2
        if c < best_cost:
            best_cost = c
            best_k = k
    gap = float(abs(float(evals_p[best_k + 1]) - float(evals_p[best_k])))
    return int(best_k), int(best_k + 1), gap


def perturbation_robustness_summary_for_batch(
    *,
    cache: PauliCache,
    seed_start: int,
    n_seeds: int,
    n_terms: int,
    perturb_terms: Optional[int],
    iso_eps: float,
    eps_neighbor: float,
    times: List[float],
    keep_mass: float,
    ent_step: float,
    leak_step: float,
    edges_coarse: np.ndarray,
    q_bins_coarse: int,
```

```
    stable_frac: float,
    eta_list: List[float],
    reps: int,
    pairs_per_seed: int,
) -> Dict[float, Dict[str, Dict[str, float]]]:
    """\
    Compute perturbation robustness summaries for both REAL and spectrum-
matched Haar NULL.

    v19 change: Robustness is quantified with a *basis-invariant* 2D
subspace overlap.

    For each base seed we construct:
      H_REAL (Pauli-sum) and its eigensystem (E, V).
      H_NULL = U diag(E) U^† where U is Haar-random (seeded) (spectrum-
matched).

    For each eta, rep we construct a small Hermitian perturbation ΔH
(Pauli-sum, seeded) and apply
    the same ΔH in the computational basis:
      H_REAL' = H_REAL + eta * ΔH
      H_NULL' = H_NULL + eta * ΔH
    Then we re-diagonalize both. Each baseline neighbor-pair manifold
(i,i+1) is tracked to the
    best-matching *neighbor* pair (k,k+1) in the perturbed spectrum by
energy proximity.

    Primary outputs per eta, per model:
      - pair_retention_rate: fraction with matched neighbor gap <
eps_neighbor
      - subspace_overlap_(mean/p10/p50/p90): overlap between baseline and
perturbed 2D subspaces
    Secondary outputs (kept for context): coarse-signature retention and
feature drifts.
    """
    d = cache.d
    perturb_terms_eff = int(perturb_terms) if perturb_terms is not None
else int(n_terms)

    # accumulators: eta -> model -> lists/sums
    out: Dict[float, Dict[str, Dict[str, object]]] = {}
    for eta in eta_list:
        out[float(eta)] = {
            "REAL": {
                "pairs": 0.0,
                "retained": 0.0,
                "sig_retained": 0.0,
                "d_ent": 0.0,
                "d_leak": 0.0,
                "ov_all": [],
                "ov_ret": [],
                "logR_ret": [],
                "gap_out_ret": [],
            },
            "NULL": {
                "pairs": 0.0,
```

```
                "retained": 0.0,
                "sig_retained": 0.0,
                "d_ent": 0.0,
                "d_leak": 0.0,
                "ov_all": [],
                "ov_ret": [],
                "logR_ret": [],
                "gap_out_ret": [],
            },
        }

    if not eta_list or reps <= 0 or n_seeds <= 0:
        return out

    for s in range(int(n_seeds)):
        seed = int(seed_start + s)

        # Base REAL spectrum/eigenvectors
        H_real = build_random_pauli_hamiltonian_cached(cache, n_terms,
seed)
        evals_base, evecs_real_base = np.linalg.eigh(H_real)

        # Base NULL eigenvectors (Haar basis) with same eigenvalues, and
the explicit NULL Hamiltonian
        _, evecs_null_base = null_haar_basis_eigs(evals_base, seed=seed,
tag="NULL_HAAR_BASIS")
        H_null = evecs_null_base @ np.diag(evals_base) @
evecs_null_base.conj().T
        H_null = 0.5 * (H_null + H_null.conj().T)

        pairs = find_neighbor_pairs(evals_base, eps_neighbor)
        if not pairs:
            continue

        # Precompute base candidate features for both models; optionally
select top pairs_per_seed by score.
        # We additionally store the baseline 2D subspace basis U (d x 2)
for overlap computations.
        base_lists: Dict[str,
List[Tuple[int,int,float,float,int,float,SigQGCoarse,np.ndarray]]] =
{"REAL": [], "NULL": []}
        for model, evecs in [("REAL", evecs_real_base), ("NULL",
evecs_null_base)]:
            feats:
List[Tuple[int,int,float,float,int,float,SigQGCoarse,float,np.ndarray]] =
[]
            for (i, j, _de) in pairs:
                ent_i = amplitude_entropy_bits(evecs[:, i])
                ent_j = amplitude_entropy_bits(evecs[:, j])
                leak, dom = leakage_proxy_fast(evals_base, evecs, i, j,
times, keep_mass=keep_mass)
                score = interestingness_score(dom, leak)
                sig_c = signature_key_qg_coarse(dom, ent_i, ent_j, leak,
d, edges_coarse, q_bins_coarse, leak_step)
                U = np.column_stack([evecs[:, i], evecs[:, j]])
```

```
                left_gap0 = float(abs(evals_base[i] - evals_base[i-1]))
if i > 0 else float('inf')
                right_gap0 = float(abs(evals_base[j+1] - evals_base[j]))
if (j + 1) < len(evals_base) else float('inf')
                gap_out0 = min(left_gap0, right_gap0)
                if not np.isfinite(gap_out0):
                    gap_out0 = left_gap0 if np.isfinite(left_gap0) else
(right_gap0 if np.isfinite(right_gap0) else 0.0)
                denom0 = max(float(_de), float(iso_eps))
                iso_log0 = float(math.log10(gap_out0 / denom0)) if
(gap_out0 > 0.0 and denom0 > 0.0) else float('nan')
                feats.append((i, j, ent_i, ent_j, dom, leak, sig_c,
score, gap_out0, iso_log0, U))
            feats.sort(key=lambda t: t[-2])  # lower score = more
"stable/interesting"
            take = int(max(1, min(int(pairs_per_seed), len(feats))))
            base_lists[model] = [(i, j, ent_i, ent_j, dom, leak, sig_c,
gap_out0, iso_log0, U) for (i, j, ent_i, ent_j, dom, leak, sig_c, _score,
gap_out0, iso_log0, U) in feats[:take]]

        for eta in eta_list:
            eta = float(eta)
            for rep in range(int(reps)):
                # Perturbation ΔH: deterministic Pauli-sum, applied
identically to REAL and NULL.
                seed_pert =
stable_hash_int(f"PERT|{seed}|eta{eta:.6g}|rep{rep}")
                dH = build_random_pauli_hamiltonian_cached(cache,
perturb_terms_eff, seed_pert)

                H_real_p = 0.5 * (H_real + (eta * dH) + (H_real + (eta *
dH)).conj().T)
                evals_real_p, evecs_real_p = np.linalg.eigh(H_real_p)

                H_null_p = 0.5 * (H_null + (eta * dH) + (H_null + (eta *
dH)).conj().T)
                evals_null_p, evecs_null_p = np.linalg.eigh(H_null_p)

                for model, evals_p, evecs_p, base_items in [
                    ("REAL", evals_real_p, evecs_real_p,
base_lists["REAL"]),
                    ("NULL", evals_null_p, evecs_null_p,
base_lists["NULL"]),
                ]:
                    for (i, j, ent_i0, ent_j0, _dom0, leak0, sig0,
gap_out0, iso_log0, U0) in base_items:
                        Ei = float(evals_base[i]); Ej =
float(evals_base[j])
                        k, l, gap = _best_match_neighbor_pair(evals_p,
Ei, Ej)
                        if k == l:
                            continue
                        retained = (gap < float(eps_neighbor))

                        # perturbed features
                        ent_i1 = amplitude_entropy_bits(evecs_p[:, k])
```

```
                        ent_j1 = amplitude_entropy_bits(evecs_p[:, l])
                        leak1, dom1 = leakage_proxy_fast(evals_p,
evecs_p, k, l, times, keep_mass=keep_mass)
                        sig1 = signature_key_qg_coarse(dom1, ent_i1,
ent_j1, leak1, d, edges_coarse, q_bins_coarse, leak_step)

                        U1 = np.column_stack([evecs_p[:, k], evecs_p[:,
l]])
                        ov = _subspace_overlap_2d(U0, U1)

                        acc = out[eta][model]
                        acc["pairs"] = float(acc["pairs"]) + 1.0
                        acc["retained"] = float(acc["retained"]) + (1.0
if retained else 0.0)
                        acc["sig_retained"] = float(acc["sig_retained"])
+ (1.0 if (sig1 == sig0) else 0.0)
                        acc["d_ent"] = float(acc["d_ent"]) + 0.5 *
(abs(ent_i1 - ent_i0) + abs(ent_j1 - ent_j0))
                        acc["d_leak"] = float(acc["d_leak"]) + abs(leak1
- leak0)
                        acc["ov_all"].append(float(ov))
                        if retained:
                            acc["ov_ret"].append(float(ov))
                            acc["logR_ret"].append(float(iso_log0))
                            acc["gap_out_ret"].append(float(gap_out0))

    # finalize to rates/means + overlap quantiles
    def _q(vals: List[float], q: float) -> float:
        if not vals:
            return 0.0
        return float(np.quantile(np.array(vals, dtype=float), q))

    for eta in eta_list:
        eta = float(eta)
        for model in ["REAL", "NULL"]:
            acc = out[eta][model]
            n = max(1.0, float(acc["pairs"]))
            acc["pair_retention_rate"] = float(acc["retained"]) / n
            acc["sig_coarse_retention_rate"] = float(acc["sig_retained"])
/ n
            acc["mean_abs_d_entropy_bits"] = float(acc["d_ent"]) / n
            acc["mean_abs_d_leak"] = float(acc["d_leak"]) / n
            acc["pairs_total"] = float(acc["pairs"])

            ov_all = list(acc["ov_all"])
            ov_ret = list(acc["ov_ret"])
            acc["subspace_overlap_all_mean"] = float(np.mean(ov_all)) if
ov_all else 0.0
            acc["subspace_overlap_all_p10"] = _q(ov_all, 0.10)
            acc["subspace_overlap_all_p50"] = _q(ov_all, 0.50)
            acc["subspace_overlap_all_p90"] = _q(ov_all, 0.90)

            acc["subspace_overlap_retained_mean"] =
float(np.mean(ov_ret)) if ov_ret else 0.0
            acc["subspace_overlap_retained_p10"] = _q(ov_ret, 0.10)
            acc["subspace_overlap_retained_p50"] = _q(ov_ret, 0.50)
```

```
            acc["subspace_overlap_retained_p90"] = _q(ov_ret, 0.90)
            acc["retained_pairs_total"] = float(len(ov_ret))

            # remove raw sums/lists to keep output clean

            logR_ret = [float(x) for x in acc.get("logR_ret", []) if
np.isfinite(x)]
            gap_out_ret = [float(x) for x in acc.get("gap_out_ret", [])
if np.isfinite(x)]
            acc["iso_log_ret_mean"] = float(np.mean(logR_ret)) if
logR_ret else float('nan')
            acc["iso_log_ret_p10"] = _q(logR_ret, 0.10) if logR_ret else
float('nan')
            acc["iso_log_ret_p50"] = _q(logR_ret, 0.50) if logR_ret else
float('nan')
            acc["iso_log_ret_p90"] = _q(logR_ret, 0.90) if logR_ret else
float('nan')

            # logR quartile table on retained subset: overlap statistics
by logR quartile
            _pairs_lr_ov = [(float(lr), float(ov)) for lr, ov in
zip(acc.get("logR_ret", []), ov_ret) if np.isfinite(lr) and
np.isfinite(ov)]
            edges_q = [float("nan"), float("nan"), float("nan")]
            quart_rows = []
            if len(_pairs_lr_ov) >= 8:
                lrs = np.array([p[0] for p in _pairs_lr_ov], dtype=float)
                ovs = np.array([p[1] for p in _pairs_lr_ov], dtype=float)
                q25, q50, q75 = np.quantile(lrs, [0.25, 0.5, 0.75])
                edges_q = [float(q25), float(q50), float(q75)]
                bins = [(-np.inf, q25), (q25, q50), (q50, q75), (q75,
np.inf)]
                labels = ["Q1", "Q2", "Q3", "Q4"]
                for (lo, hi), lab in zip(bins, labels):
                    if lo == -np.inf:
                        msk = (lrs <= hi)
                    else:
                        msk = (lrs > lo) & (lrs <= hi)
                    ov_bin = ovs[msk]
                    if ov_bin.size == 0:
                        quart_rows.append({"q": lab, "n": 0, "ov_p10":
float("nan"), "ov_p50": float("nan"), "ov_p90": float("nan")})
                    else:
                        quart_rows.append({
                            "q": lab,
                            "n": int(ov_bin.size),
                            "ov_p10": float(np.quantile(ov_bin, 0.10)),
                            "ov_p50": float(np.quantile(ov_bin, 0.50)),
                            "ov_p90": float(np.quantile(ov_bin, 0.90)),
                        })
            acc["logR_quartile_edges"] = edges_q
            acc["logR_quartiles_ret"] = quart_rows

            def _corr(a: List[float], b: List[float]) -> float:
                if len(a) < 2 or len(b) < 2 or len(a) != len(b):
                    return float('nan')
```

```
                aa = np.array(a, dtype=float)
                bb = np.array(b, dtype=float)
                m = np.isfinite(aa) & np.isfinite(bb)
                if int(np.sum(m)) < 2:
                    return float('nan')
                return float(np.corrcoef(aa[m], bb[m])[0, 1])

            # correlations on the retained subset (aligned pairs)
            _lr_pairs = [(float(lr), float(ov)) for lr, ov in
zip(acc.get("logR_ret", []), ov_ret) if np.isfinite(lr) and
np.isfinite(ov)]
            _go_pairs = [(float(go), float(ov)) for go, ov in
zip(acc.get("gap_out_ret", []), ov_ret) if np.isfinite(go) and
np.isfinite(ov)]
            acc["corr_iso_log_ov_ret"] = _corr([p[0] for p in _lr_pairs],
[p[1] for p in _lr_pairs]) if _lr_pairs else float("nan")
            acc["corr_gap_out_ov_ret"] = _corr([p[0] for p in _go_pairs],
[p[1] for p in _go_pairs]) if _go_pairs else float("nan")

            # remove raw sums/lists to keep output clean
            for k in ["pairs", "retained", "sig_retained", "d_ent",
"d_leak", "ov_all", "ov_ret"]:
                acc.pop(k, None)

    # type-ignore: nested dict contains floats only after finalization
    return out  # type: ignore[return-value]




def family_rows(
    sigs: List[Tuple[int, int, int, int]],
    stable_mask: np.ndarray,
    alpha: float,
) -> Tuple[List[FamRow], Dict[Tuple[int, int, int, int], Tuple[int,
int]], float]:
    overall = Counter(sigs)
    stable = Counter([sigs[i] for i in range(len(sigs)) if
stable_mask[i]])

    n_all = len(sigs)
    n_stable = int(np.sum(stable_mask))
    stable_rate = n_stable / max(1, n_all)

    K = max(1, len(overall))
    rows: List[FamRow] = []
    counts: Dict[Tuple[int, int, int, int], Tuple[int, int]] = {}

    for sig, o in overall.items():
        st = stable.get(sig, 0)
        counts[sig] = (int(o), int(st))

        p_all = (o + alpha) / (n_all + alpha * K)
        p_st = (st + alpha) / (n_stable + alpha * K)
        enr = (p_st / p_all) / max(1e-12, stable_rate)
```

```
        p_tail = binom_tail_ge(int(st), int(o), stable_rate) if o > 0
else 1.0
        p_tail = max(1e-300, min(1.0, float(p_tail)))
        neglog10 = -math.log10(p_tail)

        rows.append(
            FamRow(
                sig=sig,
                overall=int(o),
                stable=int(st),
                expected=float(o) * stable_rate,
                p_tail=float(p_tail),
                neglog10_p=float(neglog10),
                enrichment=float(enr),
            )
        )

    rows.sort(key=lambda r: (-r.neglog10_p, -r.enrichment, -r.stable, -
r.overall))
    return rows, counts, stable_rate


def top_k_families(rows: List[FamRow], k: int, min_overall: int,
min_stable: int, p_tail_max: Optional[float]) -> List[FamRow]:
    out: List[FamRow] = []
    for r in rows:
        if r.overall >= min_overall and r.stable >= min_stable:
            if p_tail_max is None or r.p_tail <= float(p_tail_max):
                out.append(r)
        if len(out) >= k:
            break
    return out


def summarize_entropy_effect(ent_vals: np.ndarray, stable_mask:
np.ndarray, rng: np.random.Generator, B: int = 200) -> Tuple[float,
Tuple[float, float]]:
    all_med = float(np.median(ent_vals))
    st_med = float(np.median(ent_vals[stable_mask])) if
np.any(stable_mask) else all_med
    eff = float(st_med - all_med)

    n = ent_vals.size
    idx_all = np.arange(n)
    effects = []
    for _ in range(int(B)):
        samp = rng.choice(idx_all, size=n, replace=True)
        samp_vals = ent_vals[samp]
        samp_mask = stable_mask[samp]
        all_m = float(np.median(samp_vals))
        st_m = float(np.median(samp_vals[samp_mask])) if
np.any(samp_mask) else all_m
        effects.append(st_m - all_m)
    lo, hi = np.quantile(np.array(effects, dtype=float), [0.025, 0.975])
    return eff, (float(lo), float(hi))
```

```
def cohens_d(x: np.ndarray, y: np.ndarray) -> float:
    x = np.asarray(x, dtype=float)
    y = np.asarray(y, dtype=float)
    if x.size < 2 or y.size < 2:
        return 0.0
    mx, my = float(np.mean(x)), float(np.mean(y))
    vx, vy = float(np.var(x, ddof=1)), float(np.var(y, ddof=1))
    pooled = math.sqrt(max(1e-12, ((x.size - 1) * vx + (y.size - 1) * vy)
/ max(1, (x.size + y.size - 2))))
    return float((mx - my) / pooled)


def stable_mask_from_scores_and_leak(
    scores: np.ndarray,
    leaks: np.ndarray,
    stable_frac: float,
    stable_leak_max: Optional[float],
    stable_leak_quantile: Optional[float],
) -> np.ndarray:
    """
    Fixed-fraction stable selection per model (v15):

    Select exactly k = round(stable_frac * n) (minimum 1 if n>0) items
per model,
    based on score (lower is better). Optionally impose a leakage
constraint:

      - absolute: leak <= stable_leak_max
      - or quantile: leak <= quantile(leak, stable_leak_quantile)

    If a leakage constraint is provided:
      - If eligible (leak <= threshold) count >= k: pick the best k among
eligible by score.
      - If eligible count < k: pick all eligible, then fill remainder
from ineligible by (score, leak).
    """
    n = int(scores.size)
    if n == 0:
        return np.zeros((0,), dtype=bool)

    stable_frac = float(stable_frac)
    stable_frac = max(0.0, min(1.0, stable_frac))
    k = int(round(stable_frac * n))
    k = max(1, min(n, k))

    order = np.lexsort((leaks, scores))  # score primary, leak secondary

    if stable_leak_max is None and stable_leak_quantile is None:
        chosen = order[:k]
        mask = np.zeros((n,), dtype=bool)
        mask[chosen] = True
        return mask

    if stable_leak_max is not None:
        leak_thr = float(stable_leak_max)
```

```
    else:
        q = float(stable_leak_quantile) if stable_leak_quantile is not
None else 1.0
        q = max(0.0, min(1.0, q))
        leak_thr = float(np.quantile(leaks, q))

    eligible = leaks <= leak_thr
    eligible_order = [i for i in order if eligible[i]]

    chosen: List[int] = []
    if len(eligible_order) >= k:
        chosen = eligible_order[:k]
    else:
        chosen = eligible_order[:]
        need = k - len(chosen)
        if need > 0:
            for i in order:
                if eligible[i]:
                    continue
                chosen.append(i)
                need -= 1
                if need == 0:
                    break

    mask = np.zeros((n,), dtype=bool)
    mask[chosen] = True
    return mask


# ----------------------------
# Batch runner (paired REAL + NULL)
# ----------------------------

@dataclass
class BatchResult:
    batch_id: int
    seed_offset: int
    model: str
    n_candidates: int
    stable_rate: float
    stable_rate_scoreonly: float
    score_stats: Tuple[float, float, float]
    leak_stats: Tuple[float, float, float]
    ent_stats: Tuple[float, float, float]
    dom_stats: Tuple[float, float, float]  # mean, median, max
    gap_in_stats: Tuple[float, float, float]      # mean, median, p90
    gap_in_cond_stats: Tuple[float, float, float] # conditional on ΔE_in
> iso_eps
    gap_in_cond_n: int
    gap_out_stats: Tuple[float, float, float]     # mean, median, p90
    iso_log_stats: Tuple[float, float, float]     # mean, median, p90 of
log10(ΔE_out/max(ΔE_in, eps))
    ent_pool: np.ndarray
    leak_vals: np.ndarray
    dom_vals: np.ndarray
    top_qg: List[FamRow]
```

```
    top_qg_coarse: List[FamRow]
    effect_entropy_bits: float
    effect_ci: Tuple[float, float]
    top_keys_qg: List[SigQGlobal]
    top_keys_qg_coarse: List[SigQGCoarse]
    counts_qg: Dict[SigQGlobal, Tuple[int, int]]
    counts_qg_coarse: Dict[SigQGCoarse, Tuple[int, int]]


def compute_batch_result(
    *,
    batch_id: int,
    seed_offset: int,
    model: str,
    cands: List[Candidate],
    stable_frac: float,
    stable_leak_max: Optional[float],
    stable_leak_quantile: Optional[float],
    topK: int,
    min_overall: int,
    min_stable: int,
    alpha: float,
    p_tail_max: Optional[float],
    bootstrap: int,
    iso_eps: float,
) -> BatchResult:
    if not cands:
        return BatchResult(
            batch_id=batch_id, seed_offset=seed_offset, model=model,
            n_candidates=0,
            stable_rate=0.0, stable_rate_scoreonly=0.0,
            score_stats=(0.0, 0.0, 0.0),
            leak_stats=(0.0, 0.0, 0.0),
            ent_stats=(0.0, 0.0, 0.0),
            dom_stats=(0.0, 0.0, 0.0),
            gap_in_stats=(0.0, 0.0, 0.0),
            gap_in_cond_stats=(0.0, 0.0, 0.0), gap_in_cond_n=0,
            gap_out_stats=(0.0, 0.0, 0.0),
            iso_log_stats=(0.0, 0.0, 0.0),
            ent_pool=np.array([], dtype=float),
            leak_vals=np.array([], dtype=float),
            dom_vals=np.array([], dtype=float),
            top_qg=[], top_qg_coarse=[],
            effect_entropy_bits=0.0, effect_ci=(0.0, 0.0),
            top_keys_qg=[], top_keys_qg_coarse=[],
            counts_qg={}, counts_qg_coarse={},
        )

    scores = np.array([c.score for c in cands], dtype=float)
    leaks = np.array([c.leak for c in cands], dtype=float)
    doms = np.array([c.dom for c in cands], dtype=float)
    ent_pool = np.array([c.ent_i for c in cands] + [c.ent_j for c in
cands], dtype=float)

    gap_in = np.array([c.delta_e for c in cands], dtype=float)
    gap_out = np.array([c.gap_out for c in cands], dtype=float)
```

```
    iso_log = np.array([c.iso_log for c in cands], dtype=float)
    # Robust statistics for gaps/isolations (finite-only)
    fin = np.isfinite(gap_in) & np.isfinite(gap_out) &
np.isfinite(iso_log)
    gap_in_f = gap_in[fin]
    gap_out_f = gap_out[fin]
    iso_log_f = iso_log[fin]
    def _stats_mean_med_p90(a: np.ndarray) -> Tuple[float, float, float]:
        if a.size == 0:
            return (0.0, 0.0, 0.0)
        return (float(np.mean(a)), float(np.median(a)),
float(np.quantile(a, 0.9)))
    gap_in_stats = _stats_mean_med_p90(gap_in_f)
    gap_out_stats = _stats_mean_med_p90(gap_out_f)
    iso_log_stats = _stats_mean_med_p90(iso_log_f)
    cond = gap_in_f > float(iso_eps)
    gap_in_cond = gap_in_f[cond]
    gap_in_cond_n = int(gap_in_cond.size)
    gap_in_cond_stats = _stats_mean_med_p90(gap_in_cond)

    mask_scoreonly = stable_mask_from_scores_and_leak(scores, leaks,
stable_frac, None, None)
    stable_rate_scoreonly = float(np.sum(mask_scoreonly) / max(1,
len(cands)))

    mask = stable_mask_from_scores_and_leak(scores, leaks, stable_frac,
stable_leak_max, stable_leak_quantile)

    sigs_qg = [c.sig_qg for c in cands]
    rows_qg, counts_qg, stable_rate_qg = family_rows(sigs_qg, mask,
alpha=alpha)
    top_qg = top_k_families(rows_qg, k=topK, min_overall=min_overall,
min_stable=min_stable, p_tail_max=p_tail_max)

    sigs_qg_c = [c.sig_qg_coarse for c in cands]
    rows_qg_c, counts_qg_c, _ = family_rows(sigs_qg_c, mask, alpha=alpha)
    top_qg_c = top_k_families(rows_qg_c, k=topK, min_overall=min_overall,
min_stable=min_stable, p_tail_max=p_tail_max)

    rng_eff =
rng_from_seed(stable_hash_int(f"{model}|batch{batch_id}|eff"))
    eff, ci = summarize_entropy_effect(ent_pool, np.repeat(mask, 2),
rng_eff, B=bootstrap)

    return BatchResult(
        batch_id=batch_id, seed_offset=seed_offset, model=model,
        n_candidates=len(cands),
        stable_rate=float(stable_rate_qg),
        stable_rate_scoreonly=float(stable_rate_scoreonly),
        score_stats=(float(scores.mean()), float(np.median(scores)),
float(scores.max())),
        leak_stats=(float(leaks.mean()), float(np.median(leaks)),
float(leaks.min())),
        ent_stats=(float(ent_pool.mean()), float(np.median(ent_pool)),
float(ent_pool.max())),
```

```python
        dom_stats=(float(doms.mean()), float(np.median(doms)),
float(doms.max())),
        gap_in_stats=gap_in_stats,
        gap_in_cond_stats=gap_in_cond_stats, gap_in_cond_n=gap_in_cond_n,
        gap_out_stats=gap_out_stats,
        iso_log_stats=iso_log_stats,
        ent_pool=ent_pool,
        leak_vals=leaks,
        dom_vals=doms,
        top_qg=top_qg,
        top_qg_coarse=top_qg_c,
        effect_entropy_bits=eff,
        effect_ci=ci,
        top_keys_qg=[r.sig for r in top_qg],
        top_keys_qg_coarse=[r.sig for r in top_qg_c],
        counts_qg=counts_qg,
        counts_qg_coarse=counts_qg_c,
    )


def run_paired_batches(
    *,
    cache: PauliCache,
    n_terms: int,
    seeds_per_batch: int,
    n_batches: int,
    base_seed: int,
    batch_stride: int,
    eps_neighbor: float,
    ent_step: float,
    leak_step: float,
    times: List[float],
    keep_mass: float,
    iso_eps: float,
    stable_frac: float,
    stable_leak_max: Optional[float],
    stable_leak_quantile: Optional[float],
    topK: int,
    min_overall: int,
    min_stable: int,
    alpha: float,
    q_bins: int,
    q_bins_coarse: int,
    p_tail_max: Optional[float],
    bootstrap: int,
    perturb_eta: List[float],
    perturb_reps: int,
    perturb_seeds: int,
    perturb_terms: Optional[int],
    perturb_pairs_per_seed: int,
    # v22 open-system options
    open_system: bool,
    os_include_baselines: bool,
    os_pairs_per_batch: int,
    os_noise_model: str,
    os_gamma_phi: float,
```

```
    os_gamma_1: float,
    os_t_max: float,
    os_t_steps: int,
    os_dt_internal: Optional[float],
    os_states_mode: str,
    os_use_stable_pool: bool,
    os_logical_basis: str,
    os_noise_qubits: str,
    os_noise_subset: Optional[List[int]],
    os_report_quantiles: Tuple[float, float, float],
) -> Tuple[List[BatchResult], List[BatchResult], List[Dict[str, float]],
List[Dict[float, Dict[str, Dict[str, float]]]], List[Dict[str, Dict[str,
object]]]]:
    real_results: List[BatchResult] = []
    null_results: List[BatchResult] = []
    scoreboards: List[Dict[str, float]] = []
    perturb_summaries: List[Dict[float, Dict[str, Dict[str, float]]]] =
[]
    open_system_summaries: List[Dict[str, Dict[str, object]]] = []

    d = cache.d

    for b in range(n_batches):
        offset = base_seed + b * batch_stride
        t0 = time.time()

        real_cands: List[Candidate] = []
        null_cands: List[Candidate] = []

        for s in range(seeds_per_batch):
            seed = offset + s
            real_cands.extend(
                generate_candidates_for_seed(
                    model="REAL",
                    seed=seed,
                    cache=cache,
                    n_terms=n_terms,
                    eps_neighbor=eps_neighbor,
                    ent_step=ent_step,
                    leak_step=leak_step,
                    times=times,
                    keep_mass=keep_mass,
                    iso_eps=iso_eps,
                )
            )
            null_cands.extend(
                generate_candidates_for_seed(
                    model="NULL_HAAR_BASIS",
                    seed=seed,
                    cache=cache,
                    n_terms=n_terms,
                    eps_neighbor=eps_neighbor,
                    ent_step=ent_step,
                    leak_step=leak_step,
                    times=times,
                    keep_mass=keep_mass,
```

```
                iso_eps=iso_eps,
            )
        )

    pooled_ent = np.array(
        [c.ent_i for c in real_cands] + [c.ent_j for c in real_cands]
+
        [c.ent_i for c in null_cands] + [c.ent_j for c in
null_cands],
        dtype=float
    )

    edges_fine = make_quantile_edges(pooled_ent, q_bins=q_bins) if
pooled_ent.size else make_quantile_edges(np.array([0.0]), q_bins=q_bins)
    edges_coarse = make_quantile_edges(pooled_ent,
q_bins=q_bins_coarse) if pooled_ent.size else
make_quantile_edges(np.array([0.0]), q_bins=q_bins_coarse)

    real_cands = assign_quantile_signatures(real_cands, d,
edges_fine, q_bins, edges_coarse, q_bins_coarse, leak_step)
    null_cands = assign_quantile_signatures(null_cands, d,
edges_fine, q_bins, edges_coarse, q_bins_coarse, leak_step)

    # v19: perturbation robustness (optional; uses only a small
prefix of seeds to control runtime)
    pert_summary: Dict[float, Dict[str, Dict[str, float]]] = {}
    if perturb_eta and int(perturb_seeds) > 0:
        seed_start = int(offset)
        n_use = int(min(int(perturb_seeds), int(seeds_per_batch)))
        pert_summary = perturbation_robustness_summary_for_batch(
            cache=cache,
            seed_start=seed_start,
            n_seeds=n_use,
            n_terms=n_terms,
            perturb_terms=perturb_terms,
            iso_eps=iso_eps,
            eps_neighbor=eps_neighbor,
            times=times,
            keep_mass=keep_mass,
            ent_step=ent_step,
            leak_step=leak_step,
            edges_coarse=edges_coarse,
            q_bins_coarse=q_bins_coarse,
            stable_frac=stable_frac,
            eta_list=list(perturb_eta),
            reps=int(perturb_reps),
            pairs_per_seed=int(perturb_pairs_per_seed),
        )
    perturb_summaries.append(pert_summary)

    # v22: open-system (Lindblad) evaluation (optional)
    os_summary: Dict[str, Dict[str, object]] = {}
    if bool(open_system):
        os_summary = open_system_summary_for_batch(
            include_baselines=bool(os_include_baselines),
            cache=cache,
```

```
                n_terms=n_terms,
                batch_id=b,
                seed_offset=offset,
                real_cands=real_cands,
                null_cands=null_cands,
                stable_frac=stable_frac,
                stable_leak_max=stable_leak_max,
                stable_leak_quantile=stable_leak_quantile,
                use_stable_pool=bool(os_use_stable_pool),
                os_pairs_per_batch=int(os_pairs_per_batch),
                noise_model=str(os_noise_model),
                gamma_phi=float(os_gamma_phi),
                gamma_1=float(os_gamma_1),
                t_max=float(os_t_max),
                t_steps=int(os_t_steps),
                dt_internal=os_dt_internal,
                states_mode=str(os_states_mode),
                q_report=tuple(os_report_quantiles),
                logical_basis=str(os_logical_basis),
                os_noise_qubits=str(os_noise_qubits),
                os_noise_subset=os_noise_subset,
            )
        open_system_summaries.append(os_summary)


        elapsed = time.time() - t0
        print(f"Batch {b+1}/{n_batches} generated: REAL={len(real_cands)}
NULL={len(null_cands)} (elapsed {elapsed:.1f}s)")

        R = compute_batch_result(
            batch_id=b,
            seed_offset=offset,
            model="REAL",
            cands=real_cands,
            stable_frac=stable_frac,
            stable_leak_max=stable_leak_max,
            stable_leak_quantile=stable_leak_quantile,
            topK=topK,
            min_overall=min_overall,
            min_stable=min_stable,
            alpha=alpha,
            p_tail_max=p_tail_max,
            bootstrap=bootstrap,
            iso_eps=iso_eps,
        )
        N = compute_batch_result(
            batch_id=b,
            seed_offset=offset,
            model="NULL_HAAR_BASIS",
            cands=null_cands,
            stable_frac=stable_frac,
            stable_leak_max=stable_leak_max,
            stable_leak_quantile=stable_leak_quantile,
            topK=topK,
            min_overall=min_overall,
            min_stable=min_stable,
```

```
            alpha=alpha,
            p_tail_max=p_tail_max,
            bootstrap=bootstrap,
            iso_eps=iso_eps,
        )

        real_results.append(R)
        null_results.append(N)

        sb = {}
        sb["delta_median_entropy_bits"] = safe_median(R.ent_pool) -
safe_median(N.ent_pool)
        sb["delta_median_leak"] = safe_median(R.leak_vals) -
safe_median(N.leak_vals)
        sb["delta_median_dom"] = safe_median(R.dom_vals) -
safe_median(N.dom_vals)
        sb["entropy_cohens_d"] = cohens_d(R.ent_pool, N.ent_pool)
        scoreboards.append(sb)

    return real_results, null_results, scoreboards, perturb_summaries,
open_system_summaries



# ----------------------------
# Open-system (Lindblad) evaluation (v22)
# ----------------------------

def op_on_qubit(n_qubits: int, op2: np.ndarray, target: int) ->
np.ndarray:
    """Embed a single-qubit operator op2 onto qubit 'target' (0-indexed,
left-to-right)."""
    mats = []
    for q in range(n_qubits):
        mats.append(op2 if q == target else PAULI_MATS["I"])
    return kron_n(mats)

def build_noise_operators(
    n_qubits: int,
    noise_model: str,
    gamma_phi: float,
    gamma_1: float,
    subset: Optional[List[int]] = None,
) -> Tuple[List[np.ndarray], List[np.ndarray]]:
    """
    Build Lindblad jump operators L_k and precompute L_k^† L_k.

    noise_model:
      - 'dephasing'  : L_k = sqrt(gamma_phi) * Z_k
      - 'amp_damp'   : L_k = sqrt(gamma_1) * sigma_minus_k
      - 'both'       : union of the above
    """
    if subset is None:
        qubits = list(range(int(n_qubits)))
    else:
        qubits = [int(x) for x in subset]
```

```
    noise_model = str(noise_model).lower().strip()
    Ls: List[np.ndarray] = []

    # Single-qubit lowering operator sigma^- = |0><1|
    sig_minus = np.array([[0.0, 1.0], [0.0, 0.0]], dtype=complex)

    if noise_model in ("dephasing", "both"):
        g = float(gamma_phi)
        if g > 0:
            for q in qubits:
                Ls.append(math.sqrt(g) * op_on_qubit(n_qubits,
PAULI_MATS["Z"], q))

    if noise_model in ("amp_damp", "amplitude_damping", "both"):
        g = float(gamma_1)
        if g > 0:
            for q in qubits:
                Ls.append(math.sqrt(g) * op_on_qubit(n_qubits, sig_minus,
q))

    if noise_model in ("depolar", "depolarizing"):
        # Simple depolarizing via Pauli jumps (not a true continuous-time
depolarizing channel,
        # but a useful stress test). We distribute the rate equally among
X,Y,Z on each qubit.
        g = float(gamma_phi)  # reuse gamma_phi as depolar rate
        if g > 0:
            per = g / 3.0
            for q in qubits:
                for p in ("X", "Y", "Z"):
                    Ls.append(math.sqrt(per) * op_on_qubit(n_qubits,
PAULI_MATS[p], q))

    LLs = [L.conj().T @ L for L in Ls]
    return Ls, LLs


def lindblad_rhs(rho: np.ndarray, H: np.ndarray, Ls: List[np.ndarray],
LLs: List[np.ndarray]) -> np.ndarray:
    """Compute d rho / dt for Lindblad master equation."""
    dr = -1j * (H @ rho - rho @ H)
    for L, LL in zip(Ls, LLs):
        dr += L @ rho @ L.conj().T - 0.5 * (LL @ rho + rho @ LL)
    return dr


def rk4_step(rho: np.ndarray, dt: float, H: np.ndarray, Ls:
List[np.ndarray], LLs: List[np.ndarray]) -> np.ndarray:
    k1 = lindblad_rhs(rho, H, Ls, LLs)
    k2 = lindblad_rhs(rho + 0.5 * dt * k1, H, Ls, LLs)
    k3 = lindblad_rhs(rho + 0.5 * dt * k2, H, Ls, LLs)
    k4 = lindblad_rhs(rho + dt * k3, H, Ls, LLs)
    out = rho + (dt / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4)
    # numerical hygiene
    out = 0.5 * (out + out.conj().T)
```

```
    tr = np.trace(out)
    if abs(tr) > 0:
        out = out / tr
    return out


def evolve_density_matrix(
    rho0: np.ndarray,
    H: np.ndarray,
    Ls: List[np.ndarray],
    LLs: List[np.ndarray],
    t_max: float,
    t_steps: int,
    dt_internal: Optional[float] = None,
) -> Tuple[np.ndarray, np.ndarray]:
    """
    Evolve rho0 from t=0..t_max over t_steps points.
    Returns (t_grid, rhos) where rhos has shape (t_steps, d, d).
    """
    t_steps = int(t_steps)
    if t_steps < 2:
        t_steps = 2
    t_grid = np.linspace(0.0, float(t_max), t_steps)
    dt = float(t_grid[1] - t_grid[0])
    if dt_internal is None:
        sub = 1
    else:
        sub = max(1, int(math.ceil(dt / float(dt_internal))))
    dt_sub = dt / sub

    d = H.shape[0]
    rhos = np.zeros((t_steps, d, d), dtype=complex)
    rho = rho0.copy()
    rho = 0.5 * (rho + rho.conj().T)
    tr = np.trace(rho)
    if abs(tr) > 0:
        rho = rho / tr
    rhos[0] = rho

    for k in range(1, t_steps):
        for _ in range(sub):
            rho = rk4_step(rho, dt_sub, H, Ls, LLs)
        rhos[k] = rho
    return t_grid, rhos


def pure_state_density(psi: np.ndarray) -> np.ndarray:
    psi = np.asarray(psi, dtype=complex).reshape(-1)
    return np.outer(psi, psi.conj())



def _fix_phase(psi: np.ndarray) -> np.ndarray:
    """Deterministic global phase fix: rotate so the largest-magnitude
component is real and non-negative."""
    psi = np.asarray(psi, dtype=complex).reshape(-1)
```

```
    k = int(np.argmax(np.abs(psi)))
    a = psi[k]
    if abs(a) < 1e-15:
        return psi
    phase = np.exp(-1j * np.angle(a))
    psi2 = psi * phase
    # enforce non-negative real for the pivot element
    if np.real(psi2[k]) < 0:
        psi2 = -psi2
    return psi2


def noise_diagonal_logical_basis(
    psi0: np.ndarray,
    psi1: np.ndarray,
    Ls: List[np.ndarray],
) -> Tuple[np.ndarray, np.ndarray]:
    """
    Choose a logical basis inside span{psi0, psi1} by diagonalizing the
projected noise intensity:
        A = Σ_k (P L_k P)† (P L_k P)  (restricted to the 2D subspace)
    Returns (psi0_new, psi1_new) as orthonormal vectors.
    """
    # Orthonormal input basis assumed (eigenvectors).
    psi0 = np.asarray(psi0, dtype=complex).reshape(-1)
    psi1 = np.asarray(psi1, dtype=complex).reshape(-1)

    # Build 2x2 matrix A in the {psi0, psi1} basis.
    A = np.zeros((2, 2), dtype=complex)
    bra0 = psi0.conj()
    bra1 = psi1.conj()

    for L in Ls:
        # J = P L P restricted to subspace basis
        Lpsi0 = L @ psi0
        Lpsi1 = L @ psi1
        j00 = np.vdot(psi0, Lpsi0)
        j01 = np.vdot(psi0, Lpsi1)
        j10 = np.vdot(psi1, Lpsi0)
        j11 = np.vdot(psi1, Lpsi1)
        J = np.array([[j00, j01], [j10, j11]], dtype=complex)
        A += J.conj().T @ J

    # Hermitize for numerical stability
    A = 0.5 * (A + A.conj().T)

    # Eigen-decompose (ascending eigenvalues => first vector is "least
noisy" in this proxy)
    w, U = np.linalg.eigh(A)

    # Compose new basis vectors in full Hilbert space
    u0 = U[:, 0]
    u1 = U[:, 1]
    psi0n = u0[0] * psi0 + u0[1] * psi1
    psi1n = u1[0] * psi0 + u1[1] * psi1
```

```
    # Normalize and fix phases deterministically
    psi0n = psi0n / max(np.linalg.norm(psi0n), 1e-15)
    psi1n = psi1n / max(np.linalg.norm(psi1n), 1e-15)
    psi0n = _fix_phase(psi0n)
    psi1n = _fix_phase(psi1n)

    # Re-orthonormalize (Gram-Schmidt) to suppress drift
    psi1n = psi1n - np.vdot(psi0n, psi1n) * psi0n
    psi1n = psi1n / max(np.linalg.norm(psi1n), 1e-15)
    psi1n = _fix_phase(psi1n)
    return psi0n, psi1n

def open_system_metrics_for_pair(
    *,
    H: np.ndarray,
    psi0: np.ndarray,
    psi1: np.ndarray,
    Ls: List[np.ndarray],
    LLs: List[np.ndarray],
    t_max: float,
    t_steps: int,
    dt_internal: Optional[float],
    states_mode: str,
    probe_seed: Optional[int] = None,
) -> Dict[str, np.ndarray]:
    """
    Compute leakage and fidelities over time for a single 2D subspace
spanned by psi0, psi1.

    Returns dict with arrays of length t_steps:
      - leak_mean
      - fid_uncond_mean
      - fid_cond_mean
    """
    # Projector onto the 2D subspace
    P = np.outer(psi0, psi0.conj()) + np.outer(psi1, psi1.conj())

    # logical test states
    s0 = psi0
    s1 = psi1
    test_states = [s0, s1]

    mode = str(states_mode).lower().strip()

    if mode in ("zx", "z+x", "full", "4"):
        sp = (s0 + s1) / math.sqrt(2.0)
        sip = (s0 + 1j * s1) / math.sqrt(2.0)
        test_states += [sp, sip]
    elif mode.startswith("rand"):
        # K random manifold-internal probes (Bloch-uniform in
span{s0,s1})
        k_str = mode[4:]
        K = int(k_str) if k_str.isdigit() else 32
        K = max(0, int(K))
        if K > 0:
            seed = int(probe_seed) if probe_seed is not None else 0
```

```
            rng =
np.random.default_rng(stable_hash_int(f"OS_PROBE|{seed}|K{K}"))
            us = rng.random(K)
            vs = rng.random(K)
            thetas = np.arccos(1.0 - 2.0 * us)
            phis = 2.0 * np.pi * vs
            for th, ph in zip(thetas, phis):
                psi = (math.cos(float(th) / 2.0) * s0) + (np.exp(1j *
float(ph)) * math.sin(float(th) / 2.0) * s1)
                psi = psi / max(np.linalg.norm(psi), 1e-15)
                test_states.append(psi)

    t_grid = None
    leak_list = []
    fu_list = []
    fc_list = []

    for psi in test_states:
        rho0 = pure_state_density(psi)
        t_grid, rhos = evolve_density_matrix(
            rho0=rho0,
            H=H,
            Ls=Ls,
            LLs=LLs,
            t_max=t_max,
            t_steps=t_steps,
            dt_internal=dt_internal,
        )

        # in-subspace weight
        w = np.real(np.trace(P @ rhos, axis1=1, axis2=2))
        w = np.clip(w, 0.0, 1.0)
        leak = 1.0 - w

        # unconditional fidelity: <psi|rho|psi>
        # (psi is in the code subspace at t=0; leakage reduces this
expectation naturally)
        fu = np.real(np.einsum("i,tij,j->t", psi.conj(), rhos, psi))
        fu = np.clip(fu, 0.0, 1.0)

        # conditional: <psi|rho|psi> / Tr(P rho)
        denom = np.clip(w, 1e-15, 1.0)
        fc = fu / denom
        fc = np.clip(fc, 0.0, 1.0)

        leak_list.append(leak)
        fu_list.append(fu)
        fc_list.append(fc)

    leak_mean = np.mean(np.stack(leak_list, axis=0), axis=0)
    fu_mean = np.mean(np.stack(fu_list, axis=0), axis=0)
    fc_mean = np.mean(np.stack(fc_list, axis=0), axis=0)

    return {
        "t": t_grid,
        "leak_mean": leak_mean,
```

```
        "fid_uncond_mean": fu_mean,
        "fid_cond_mean": fc_mean,
    }


def first_crossing_time(t: np.ndarray, y: np.ndarray, thresh: float,
direction: str) -> float:
    """
    Return first time where y crosses thresh.
      - direction='below': first t where y < thresh
      - direction='above': first t where y > thresh
    If no crossing, return t[-1].
    """
    direction = str(direction).lower().strip()
    if direction == "below":
        idx = np.where(y < float(thresh))[0]
    else:
        idx = np.where(y > float(thresh))[0]
    if idx.size == 0:
        return float(t[-1])
    return float(t[int(idx[0])])


def summarize_timeseries_across_pairs(
    t: np.ndarray,
    curves: np.ndarray,
    q: Tuple[float, float, float] = (0.1, 0.5, 0.9),
) -> Dict[str, np.ndarray]:
    """
    curves shape: (n_pairs, t_steps). Returns quantiles per time and
median AUC.
    """
    q = tuple(float(x) for x in q)
    qvals = np.quantile(curves, q, axis=0)
    auc = np.trapz(curves, t, axis=1)
    return {
        "q_lo": qvals[0],
        "q_med": qvals[1],
        "q_hi": qvals[2],
        "auc_med": np.median(auc) if auc.size else float("nan"),
    }


def sample_candidates_by_iso_quartile(
    cands: List[Candidate],
    which: str,
    n: int,
    rng: np.random.Generator,
) -> List[Candidate]:
    """
    which: 'Q1' or 'Q4' based on iso_log quartiles within cands.
    """
    if not cands:
        return []
    vals = np.array([c.iso_log for c in cands], dtype=float)
    q25 = float(np.quantile(vals, 0.25))
```

```
    q75 = float(np.quantile(vals, 0.75))
    if which.upper() == "Q1":
        pool = [c for c in cands if float(c.iso_log) <= q25]
    else:
        pool = [c for c in cands if float(c.iso_log) >= q75]
    if not pool:
        pool = list(cands)
    if n >= len(pool):
        return pool
    idx = rng.choice(len(pool), size=int(n), replace=False)
    return [pool[int(i)] for i in idx]


def open_system_summary_for_batch(
    *,
    include_baselines: bool,
    cache: PauliCache,
    n_terms: int,
    batch_id: int,
    seed_offset: int,
    real_cands: List[Candidate],
    null_cands: List[Candidate],
    stable_frac: float,
    stable_leak_max: Optional[float],
    stable_leak_quantile: Optional[float],
    use_stable_pool: bool,
    os_pairs_per_batch: int,
    noise_model: str,
    gamma_phi: float,
    gamma_1: float,
    t_max: float,
    t_steps: int,
    dt_internal: Optional[float],
    states_mode: str,
    q_report: Tuple[float, float, float],
    logical_basis: str,
    os_noise_qubits: str,
    os_noise_subset: Optional[List[int]],
) -> Dict[str, Dict[str, object]]:
    """
    Returns dict with keys 'REAL_Q4', 'REAL_Q1', 'NULL_Q4' and per-key
summaries.
    """
    rng =
rng_from_seed(stable_hash_int(f"OPEN_SYSTEM|{seed_offset}|{batch_id}"))

    # optionally restrict to stable pool to focus on physically relevant
candidates
    def maybe_stable_pool(cands: List[Candidate]) -> List[Candidate]:
        if (not use_stable_pool) or (not cands):
            return cands
        scores = np.array([c.score for c in cands], dtype=float)
        leaks = np.array([c.leak for c in cands], dtype=float)
        m = stable_mask_from_scores_and_leak(scores, leaks, stable_frac,
stable_leak_max, stable_leak_quantile)
        return [c for c, keep in zip(cands, m) if bool(keep)]
```

```
    real_pool = maybe_stable_pool(real_cands)
    null_pool = maybe_stable_pool(null_cands)

    # noise ops constant for this n_qubits
    subset = None
    if str(os_noise_qubits).lower().strip() == 'subset':
        if os_noise_subset is None or len(os_noise_subset) == 0:
            subset = [0]
        else:
            subset = [int(x) for x in os_noise_subset]
    Ls, LLs = build_noise_operators(cache.n_qubits, noise_model,
gamma_phi, gamma_1, subset)

    def eval_group(model: str, which: str, pool: List[Candidate]) ->
Dict[str, object]:
        samp = sample_candidates_by_iso_quartile(pool, which,
int(os_pairs_per_batch), rng)
        if not samp:
            return {"n_pairs": 0}

        # compute curves per pair (mean across logical probe states)
        leak_curves = []
        fu_curves = []
        fc_curves = []
        t_ref = None

        t_fid90 = []
        t_fid50 = []
        t_leak10 = []

        for c in samp:
            # rebuild H and eigenvectors deterministically
            H_real = build_random_pauli_hamiltonian_cached(cache,
n_terms, c.seed)
            if model == "REAL":
                H = H_real
            else:
                evals_real, _ = np.linalg.eigh(H_real)
                H = null_haar_basis_hamiltonian(evals_real, seed=c.seed)

            evals, evecs = np.linalg.eigh(H)
            psi0 = evecs[:, int(c.i)]
            psi1 = evecs[:, int(c.j)]

            if str(logical_basis).lower().strip() == 'noise_diag':
                psi0, psi1 = noise_diagonal_logical_basis(psi0, psi1, Ls)

            met = open_system_metrics_for_pair(
                H=H,
                psi0=psi0,
                psi1=psi1,
                Ls=Ls,
                LLs=LLs,
                t_max=t_max,
                t_steps=t_steps,
```

```
                dt_internal=dt_internal,
                states_mode=states_mode,

probe_seed=stable_hash_int(f"OS_PROBE|b{batch_id}|off{seed_offset}|{model
}|seed{int(c.seed)}|i{int(c.i)}|j{int(c.j)}|{states_mode}"),
            )
            t = met["t"]
            if t_ref is None:
                t_ref = t
            leak = met["leak_mean"]
            fu = met["fid_uncond_mean"]
            fc = met["fid_cond_mean"]

            leak_curves.append(leak)
            fu_curves.append(fu)
            fc_curves.append(fc)

            t_fid90.append(first_crossing_time(t, fu, 0.9, "below"))
            t_fid50.append(first_crossing_time(t, fu, 0.5, "below"))
            t_leak10.append(first_crossing_time(t, leak, 0.1, "above"))

        leak_curves = np.stack(leak_curves, axis=0)
        fu_curves = np.stack(fu_curves, axis=0)
        fc_curves = np.stack(fc_curves, axis=0)

        leak_sum = summarize_timeseries_across_pairs(t_ref, leak_curves,
q_report)
        fu_sum = summarize_timeseries_across_pairs(t_ref, fu_curves,
q_report)
        fc_sum = summarize_timeseries_across_pairs(t_ref, fc_curves,
q_report)

        return {
            "n_pairs": int(leak_curves.shape[0]),
            "t": t_ref,
            "leak": leak_sum,
            "fid_uncond": fu_sum,
            "fid_cond": fc_sum,
            "t_fid90_med": float(np.median(np.array(t_fid90))) if t_fid90
else float("nan"),
            "t_fid50_med": float(np.median(np.array(t_fid50))) if t_fid50
else float("nan"),
            "t_leak10_med": float(np.median(np.array(t_leak10))) if
t_leak10 else float("nan"),
        }

    out: Dict[str, Dict[str, object]] = {}
    out["REAL_Q4"] = eval_group("REAL", "Q4", real_pool)
    if bool(include_baselines):
        out["REAL_Q1"] = eval_group("REAL", "Q1", real_pool)
    out["NULL_Q4"] = eval_group("NULL", "Q4", null_pool)

    # attach meta
    out["_meta"] = {
        "noise_model": str(noise_model),
        "gamma_phi": float(gamma_phi),
```

```
        "gamma_1": float(gamma_1),
        "t_max": float(t_max),
        "t_steps": int(t_steps),
        "states_mode": str(states_mode),
        "use_stable_pool": bool(use_stable_pool),
        "os_pairs_per_batch": int(os_pairs_per_batch),
        "logical_basis": str(logical_basis),
        "os_noise_qubits": str(os_noise_qubits),
        "os_noise_subset": subset if subset is not None else None,
    }
    return out


# ----------------------------
# Reporting
# ----------------------------

def format_top(rows: List[FamRow], label: str, show: int = 10) -> str:
    lines = []
    lines.append(f"Top signature families ({label}):")
    lines.append("Format: sig=(a,b,c,d) | overall | stable | expected |
p_tail | -log10(p) | enrichment")
    for r in rows[:show]:
        lines.append(
            f"  {r.sig} | {r.overall:7d} | {r.stable:7d} |
{r.expected:9.2f} | {r.p_tail:8.2e} | {r.neglog10_p:9.2f} |
{r.enrichment:9.2f}x"
        )
    if len(rows) > show:
        lines.append(f"  ... ({len(rows)} total, showing {show})")
    return "\n".join(lines)


def main() -> None:
    ap = argparse.ArgumentParser(description="v0_7 convergence harness
v19 (v16 baseline + evidence-grade perturbation robustness; n_qubits=4
default).")
    ap.add_argument("--n_qubits", type=int, default=4)
    ap.add_argument("--n_terms", type=int, default=5)
    ap.add_argument("--seeds_per_batch", type=int, default=5000)
    ap.add_argument("--batches", type=int, default=3)
    ap.add_argument("--base_seed", type=int, default=0)
    ap.add_argument("--batch_stride", type=int, default=1000000)
    ap.add_argument("--eps_neighbor", type=float, default=0.05)

    ap.add_argument("--keep_mass", type=float, default=0.90)
    ap.add_argument("--ent_step", type=float, default=0.1)
    ap.add_argument("--leak_step", type=float, default=0.05)
    ap.add_argument("--times", type=float, nargs="+", default=[0.5, 1.0,
1.5])

    ap.add_argument("--stable_frac", type=float, default=0.01)
    ap.add_argument("--stable_leak_max", type=float, default=None)
    ap.add_argument("--stable_leak_quantile", type=float, default=None)
```

```
    # v19: perturbation robustness controls (disabled unless --
perturb_eta is provided and --perturb_seeds > 0)
    ap.add_argument("--perturb_eta", type=float, nargs="+", default=[])
    ap.add_argument("--perturb_reps", type=int, default=3)
    ap.add_argument("--perturb_seeds", type=int, default=0, help="Number
of seeds per batch used for perturbation robustness (0 disables).")
    ap.add_argument("--perturb_terms", type=int, default=None,
help="Number of Pauli terms in ΔH (default: same as --n_terms).")
    ap.add_argument("--perturb_pairs_per_seed", type=int, default=5,
help="Number of near-degenerate pairs per seed used in robustness summary
(lowest-score subset).")
    ap.add_argument("--iso_eps", type=float, default=1e-8, help="Clamp
for ΔE_in when forming log isolation ratio log10(ΔE_out/max(ΔE_in,
iso_eps)).")
    ap.add_argument("--no_iso_quartiles", action="store_true",
help="Disable logR quartile table in perturbation output.")
    ap.add_argument("--iso_quartiles_all_eta", action="store_true",
help="Print logR quartile tables for all eta values (default: only max
eta).")
    # v22: open-system (Lindblad) evaluation
    ap.add_argument("--open_system", action="store_true", help="Enable
open-system (Lindblad) logical retention test on selected 2D subspaces.")
    ap.add_argument("--os_include_baselines", action="store_true",
help="If set, evaluate REAL_Q4, REAL_Q1, and NULL_Q4 in each batch.")
    ap.add_argument("--os_pairs_per_batch", type=int, default=400,
help="Number of near-degenerate pairs sampled per category per batch for
open-system evaluation.")
    ap.add_argument("--os_noise_model", type=str, default="dephasing",
choices=["dephasing", "amp_damp", "both", "depolar"], help="Noise model
for Lindblad evolution.")
    ap.add_argument("--os_gamma_phi", type=float, default=0.01,
help="Dephasing (or depolar) rate gamma.")
    ap.add_argument("--os_gamma_1", type=float, default=0.01,
help="Amplitude damping rate gamma_1.")
    ap.add_argument("--os_t_max", type=float, default=5.0, help="Max
evolution time for open-system evaluation.")
    ap.add_argument("--os_t_steps", type=int, default=25, help="Number of
time points for open-system evaluation.")
    ap.add_argument("--os_dt_internal", type=float, default=None,
help="Optional internal RK4 step size. If None, one RK4 step per output
interval.")
    ap.add_argument("--os_states_mode", type=str, default="zx",
help="Logical probe states: 'z'=(|0_L>,|1_L>), 'zx' adds |+_L>,|i+_L>.
Also supports 'randK' for K random manifold-internal probes (e.g.,
rand32, rand64).")
    ap.add_argument("--logical_basis", type=str, default="eigen",
choices=["eigen", "noise_diag"],
                    help="Open-system: logical basis inside the 2D
subspace. 'eigen' uses the eigenpair basis; "
                         "'noise_diag' diagonalizes the projected noise
intensity A=Σ (P L_k P)†(P L_k P).")
    ap.add_argument("--os_noise_qubits", type=str, default="all",
choices=["all", "subset"],
                    help="Open-system: apply noise jumps on all qubits or
only a subset.")
```

```
    ap.add_argument("--os_noise_subset", type=int, nargs="+",
default=None,
                    help="Open-system: if --os_noise_qubits=subset, list
0-based qubit indices (e.g., 0 1). If omitted, defaults to [0].")
    ap.add_argument("--os_use_stable_pool", action="store_true",
help="Restrict open-system sampling to the stable pool (same stable
selection per model).")
    ap.add_argument("--os_report_quantiles", type=float, nargs=3,
default=[0.1, 0.5, 0.9], help="Quantiles to report for time series
summaries, e.g., 0.1 0.5 0.9.")


    ap.add_argument("--topK", type=int, default=25)
    ap.add_argument("--min_overall", type=int, default=3)
    ap.add_argument("--min_stable", type=int, default=3)
    ap.add_argument("--alpha", type=float, default=0.5)

    ap.add_argument("--q_bins", type=int, default=10)
    ap.add_argument("--q_bins_coarse", type=int, default=6)
    ap.add_argument("--p_tail_max", type=float, default=None)
    ap.add_argument("--bootstrap", type=int, default=200)

    ap.add_argument("--output", type=str,
default="v0_7_convergence_families_v23_output.txt")
    args = ap.parse_args()

    cache = PauliCache.build(args.n_qubits)
    d = cache.d

    header = []
    header.append("=== v0_7: Convergence + Baseline Calibration +
Perturbation Robustness + Gap/Isolation + Open-system (v23) ===")
    header.append(f"Qubits: {args.n_qubits} (d={d}) |
terms={args.n_terms}")
    header.append(f"Batches: {args.batches} × {args.seeds_per_batch}
seeds (base_seed={args.base_seed}, stride={args.batch_stride})")
    header.append(f"Neighbor eps={args.eps_neighbor:.3f}")
    header.append(f"Dominant set: keep_mass={args.keep_mass:.2f} (mass-
based; guarantees non-empty mask)")
    header.append(f"Bins (SigAbs): ent_step={args.ent_step:.3f} |
leak_step={args.leak_step:.3f}")
    header.append(f"Bins (SigQ_GLOBAL fine): q_bins={args.q_bins} (pooled
REAL+NULL per batch)")
    header.append(f"Bins (SigQG_COARSE):
q_bins_coarse={args.q_bins_coarse} + dom_bin(dom/d) + leak_bin_coarse
(pooled REAL+NULL per batch)")
    header.append(f"Leakage proxy times={args.times} (FAST analytic
evolution in eigenpair)")
    header.append(f"Stable selection: stable_frac={args.stable_frac:.3f}
per model (optional leak constraint max={args.stable_leak_max},
q={args.stable_leak_quantile})")
    header.append(f"Perturbation robustness: eta={args.perturb_eta} |
reps={args.perturb_reps} | seeds={args.perturb_seeds} |
pairs/seed={args.perturb_pairs_per_seed} |
dH_terms={args.perturb_terms}")
```

```
    header.append(f"Open-system (v23): enabled={bool(args.open_system)} |
include_baselines={bool(args.os_include_baselines)} |
pairs_per_cat={args.os_pairs_per_batch} |
noise={args.os_noise_model}(phi={args.os_gamma_phi},g1={args.os_gamma_1})
| t={args.os_t_max}/{args.os_t_steps} | states={args.os_states_mode} |
basis={args.logical_basis} |
noise_qubits={args.os_noise_qubits}:{args.os_noise_subset} |
stable_pool={bool(args.os_use_stable_pool)}")
    header.append(f"TopK={args.topK} | min_overall={args.min_overall} |
min_stable={args.min_stable} | alpha={args.alpha}")
    header.append(f"Optional family filter:
p_tail_max={args.p_tail_max}")
    header.append("Pauli ops: precomputed cache (excluding all-I)")
    header.append("")
    header_text = "\n".join(header)

    with open(args.output, "w", encoding="utf-8") as f:
        def out(s: str = "") -> None:
            print(s)
            f.write(s + "\n")

        out(header_text)

        real, null, scoreboards, perturb_summaries, open_system_summaries
= run_paired_batches(
            cache=cache,
            n_terms=args.n_terms,
            seeds_per_batch=args.seeds_per_batch,
            n_batches=args.batches,
            base_seed=args.base_seed,
            batch_stride=args.batch_stride,
            eps_neighbor=args.eps_neighbor,
            ent_step=args.ent_step,
            leak_step=args.leak_step,
            times=list(args.times),
            keep_mass=args.keep_mass,
    iso_eps=args.iso_eps,
            stable_frac=args.stable_frac,
            stable_leak_max=args.stable_leak_max,
            stable_leak_quantile=args.stable_leak_quantile,
            topK=args.topK,
            min_overall=args.min_overall,
            min_stable=args.min_stable,
            alpha=args.alpha,
            q_bins=args.q_bins,
            q_bins_coarse=args.q_bins_coarse,
            p_tail_max=args.p_tail_max,
            bootstrap=args.bootstrap,
            perturb_eta=list(args.perturb_eta),
            perturb_reps=args.perturb_reps,
            perturb_seeds=args.perturb_seeds,
            perturb_terms=args.perturb_terms,
            perturb_pairs_per_seed=args.perturb_pairs_per_seed,
            open_system=bool(args.open_system),
            os_include_baselines=bool(args.os_include_baselines),
            os_pairs_per_batch=int(args.os_pairs_per_batch),
```

```
            os_noise_model=str(args.os_noise_model),
            os_gamma_phi=float(args.os_gamma_phi),
            os_gamma_1=float(args.os_gamma_1),
            os_t_max=float(args.os_t_max),
            os_t_steps=int(args.os_t_steps),
            os_dt_internal=args.os_dt_internal,
            os_states_mode=str(args.os_states_mode),
            os_logical_basis=str(args.logical_basis),
            os_noise_qubits=str(args.os_noise_qubits),
            os_noise_subset=args.os_noise_subset,
            os_use_stable_pool=bool(args.os_use_stable_pool),
            os_report_quantiles=tuple(float(x) for x in
args.os_report_quantiles),
        )

        out("")
        for b in range(args.batches):
            out("---------------------------------------------")
            out(f"Batch {b+1}/{args.batches}
(seed_offset={real[b].seed_offset})")

            R = real[b]
            out(f"Model: REAL | candidates={R.n_candidates} |
stable_rate={R.stable_rate:.4f} (score-only
ref={R.stable_rate_scoreonly:.4f})")
            out(f"
score(mean/median/max)={R.score_stats[0]:.3f}/{R.score_stats[1]:.3f}/{R.s
core_stats[2]:.3f}")
            out(f"
leakage(mean/median/min)={R.leak_stats[0]:.3f}/{R.leak_stats[1]:.3f}/{R.l
eak_stats[2]:.3f}")
            out(f"
entropy(mean/median/max)={R.ent_stats[0]:.3f}/{R.ent_stats[1]:.3f}/{R.ent
_stats[2]:.3f}")
            out(f"
dom_count(mean/median/max)={R.dom_stats[0]:.2f}/{R.dom_stats[1]:.1f}/{R.d
om_stats[2]:.0f}")
            out(f"  gaps:
ΔE_in(mean/med/p90)={R.gap_in_stats[0]:.4g}/{R.gap_in_stats[1]:.4g}/{R.ga
p_in_stats[2]:.4g} |
ΔE_out(mean/med/p90)={R.gap_out_stats[0]:.4g}/{R.gap_out_stats[1]:.4g}/{R
.gap_out_stats[2]:.4g} |
logR(mean/med/p90)={R.iso_log_stats[0]:.3f}/{R.iso_log_stats[1]:.3f}/{R.i
so_log_stats[2]:.3f}")
            out(f"  gaps_cond(ΔE_in>iso_eps): n={R.gap_in_cond_n} |
ΔE_in(mean/med/p90)={R.gap_in_cond_stats[0]:.4g}/{R.gap_in_cond_stats[1]:
.4g}/{R.gap_in_cond_stats[2]:.4g}")
            out(f"  entropy effect (stable-
overall)={R.effect_entropy_bits:+.3f} bits  CI95={R.effect_ci}")
            out("")
            out(format_top(R.top_qg, "REAL, SigQ_GLOBAL (fine)"))
            out("")
            out(format_top(R.top_qg_coarse, "REAL, SigQG_COARSE"))
            out("")

            N = null[b]
```

```
            out(f"Model: NULL_HAAR_BASIS | candidates={N.n_candidates} |
stable_rate={N.stable_rate:.4f} (score-only
ref={N.stable_rate_scoreonly:.4f})")
            out(f"
score(mean/median/max)={N.score_stats[0]:.3f}/{N.score_stats[1]:.3f}/{N.s
core_stats[2]:.3f}")
            out(f"
leakage(mean/median/min)={N.leak_stats[0]:.3f}/{N.leak_stats[1]:.3f}/{N.l
eak_stats[2]:.3f}")
            out(f"
entropy(mean/median/max)={N.ent_stats[0]:.3f}/{N.ent_stats[1]:.3f}/{N.ent
_stats[2]:.3f}")
            out(f"
dom_count(mean/median/max)={N.dom_stats[0]:.2f}/{N.dom_stats[1]:.1f}/{N.d
om_stats[2]:.0f}")
            out(f"  gaps:
ΔE_in(mean/med/p90)={N.gap_in_stats[0]:.4g}/{N.gap_in_stats[1]:.4g}/{N.ga
p_in_stats[2]:.4g} |
ΔE_out(mean/med/p90)={N.gap_out_stats[0]:.4g}/{N.gap_out_stats[1]:.4g}/{N
.gap_out_stats[2]:.4g} |
logR(mean/med/p90)={N.iso_log_stats[0]:.3f}/{N.iso_log_stats[1]:.3f}/{N.i
so_log_stats[2]:.3f}")
            out(f"  gaps_cond(ΔE_in>iso_eps): n={N.gap_in_cond_n} |
ΔE_in(mean/med/p90)={N.gap_in_cond_stats[0]:.4g}/{N.gap_in_cond_stats[1]:
.4g}/{N.gap_in_cond_stats[2]:.4g}")
            out(f"  entropy effect (stable-
overall)={N.effect_entropy_bits:+.3f} bits  CI95={N.effect_ci}")
            out("")
            out(format_top(N.top_qg, "NULL, SigQ_GLOBAL (fine)"))
            out("")
            out(format_top(N.top_qg_coarse, "NULL, SigQG_COARSE"))
            out("")

            # v19: Perturbation robustness summary (optional)
            ps = perturb_summaries[b] if b < len(perturb_summaries) else
{}
            if ps:
                out("Perturbation robustness (2D subspace overlap,
evidence-grade):")
                out(f"  settings: eta={args.perturb_eta} |
reps={args.perturb_reps} | seeds_used={min(args.perturb_seeds,
args.seeds_per_batch)} | pairs/seed={args.perturb_pairs_per_seed} |
dH_terms={args.perturb_terms}")
                for eta in sorted(ps.keys()):
                    eta_f = float(eta)
                    for model in ["REAL", "NULL"]:
                        mtr = ps[eta_f][model]
                        out(
                            f"  eta={eta_f:.6g} | {model}: "

f"pair_retention={mtr['pair_retention_rate']:.3f} | "
                            f"subspace_ov_ret(mean/p10/p50/p90)="

f"{mtr['subspace_overlap_retained_mean']:.3f}/"

f"{mtr['subspace_overlap_retained_p10']:.3f}/"
```

```
f"{mtr['subspace_overlap_retained_p50']:.3f}/"
                            f"{mtr['subspace_overlap_retained_p90']:.3f}
| "

f"subspace_ov_all_mean={mtr['subspace_overlap_all_mean']:.3f} |
"f"logR_ret(p50)={mtr.get('iso_log_ret_p50', float('nan')):.2f} |
"f"corr(logR,ov)_ret={mtr.get('corr_iso_log_ov_ret', float('nan')):.2f} |
"

f"sig_coarse_ret={mtr['sig_coarse_retention_rate']:.3f} | "

f"mean|Δentropy|={mtr['mean_abs_d_entropy_bits']:.3f} bits | "
                            f"mean|Δleak|={mtr['mean_abs_d_leak']:.3f} |
"
                            f"pairs={int(mtr['pairs_total'])} |
retained_pairs={int(mtr['retained_pairs_total'])}"
                        )
                        # v21.1: logR quartile table (retained subset)
for interpretability
                        if (not args.no_iso_quartiles) and
(args.iso_quartiles_all_eta or eta_f == max(sorted(ps.keys()))):
                            qs = mtr.get("logR_quartiles_ret", [])
                            edges = mtr.get("logR_quartile_edges",
[float("nan"), float("nan"), float("nan")])
                            if qs:
                                q25s = f"{edges[0]:.3f}" if
np.isfinite(edges[0]) else "nan"
                                q50s = f"{edges[1]:.3f}" if
np.isfinite(edges[1]) else "nan"
                                q75s = f"{edges[2]:.3f}" if
np.isfinite(edges[2]) else "nan"
                                out(f"    logR quartiles (retained):
q25={q25s} q50={q50s} q75={q75s}")
                                for row in qs:
                                    p10 = row.get("ov_p10", float("nan"))
                                    p50 = row.get("ov_p50", float("nan"))
                                    p90 = row.get("ov_p90", float("nan"))
                                    p10s = f"{p10:.3f}" if
np.isfinite(p10) else "nan"
                                    p50s = f"{p50:.3f}" if
np.isfinite(p50) else "nan"
                                    p90s = f"{p90:.3f}" if
np.isfinite(p90) else "nan"
                                    out(f"      {row.get('q', 'Q?')}:
n={int(row.get('n', 0))} | ov(p10/p50/p90)={p10s}/{p50s}/{p90s}")

                out("")

            # Open-system (v23) summary (optional)
            if bool(args.open_system):
                osb = open_system_summaries[b] if b <
len(open_system_summaries) else {}
                meta = osb.get("_meta", {}) if isinstance(osb, dict) else
{}
                if isinstance(osb, dict) and meta:
```

```
                   out("Open-system logical retention (Lindblad; v23):")
                   out(
                       f"  settings: noise={meta.get('noise_model')} |
gamma_phi={meta.get('gamma_phi')} | "
                       f"gamma_1={meta.get('gamma_1')} |
t_max={meta.get('t_max')} | t_steps={meta.get('t_steps')} | "
                       f"states={meta.get('states_mode')} |
stable_pool={meta.get('use_stable_pool')} |
pairs_per_cat={meta.get('os_pairs_per_batch')}"
                   )

                   def _fmt_cat(cat: str) -> None:
                       if cat not in osb:
                           return
                       S = osb.get(cat, {})
                       n_pairs = int(S.get("n_pairs", 0)) if
isinstance(S, dict) else 0
                       if n_pairs <= 0:
                           out(f"  {cat}: n_pairs=0")
                           return
                       t = np.array(S["t"], dtype=float)
                       mid = int((t.size - 1) // 2)
                       idxs = [0, mid, int(t.size - 1)]

                       def _pt(i: int) -> str:
                           return f"t={t[i]:.3g}"

                       leak = S["leak"]
                       fu = S["fid_uncond"]
                       fc = S["fid_cond"]

                       out(f"  {cat}: n_pairs={n_pairs} |
t_fid90_med={S.get('t_fid90_med', float('nan')):.3g} |
t_leak10_med={S.get('t_leak10_med', float('nan')):.3g}")
                       out("    leak(q10/q50/q90): " + " | ".join(

f"{_pt(i)}={leak['q_lo'][i]:.3f}/{leak['q_med'][i]:.3f}/{leak['q_hi'][i]:
.3f}" for i in idxs
                       ))
                       out("    fid_uncond(q10/q50/q90): " + " | ".join(

f"{_pt(i)}={fu['q_lo'][i]:.3f}/{fu['q_med'][i]:.3f}/{fu['q_hi'][i]:.3f}"
for i in idxs
                       ))
                       out("    fid_cond(q10/q50/q90): " + " | ".join(

f"{_pt(i)}={fc['q_lo'][i]:.3f}/{fc['q_med'][i]:.3f}/{fc['q_hi'][i]:.3f}"
for i in idxs
                       ))
                       out(f"    AUC medians:
fid_uncond={fu.get('auc_med', float('nan')):.3f} |
fid_cond={fc.get('auc_med', float('nan')):.3f} |
leak={leak.get('auc_med', float('nan')):.3f}")

                   _fmt_cat("REAL_Q4")
                   _fmt_cat("REAL_Q1")
```

```
                    _fmt_cat("NULL_Q4")
                    out("")

            ov_fine = jaccard(R.top_keys_qg, N.top_keys_qg)
            ov_coarse = jaccard(R.top_keys_qg_coarse,
N.top_keys_qg_coarse)
            out(f"Batch {b+1}: Jaccard(Top-{args.topK}) REAL vs NULL:
fine={ov_fine:.3f} | coarse={ov_coarse:.3f}")
            out("")

            sb = scoreboards[b]
            out("Baseline scoreboard (REAL - NULL):")
            out(f"  delta median entropy (bits):
{sb['delta_median_entropy_bits']:+.3f}")
            out(f"  delta median leakage       :
{sb['delta_median_leak']:+.3f}")
            out(f"  delta median dom_count     :
{sb['delta_median_dom']:+.3f}")
            out(f"  entropy Cohen's d          :
{sb['entropy_cohens_d']:+.3f}")
            out("")

        out("----------------------------------------------")
        out("=== Convergence diagnostics (REAL) ===")
        real_sets_fine = [set(r.top_keys_qg) for r in real]
        real_sets_coarse = [set(r.top_keys_qg_coarse) for r in real]
        for i in range(len(real_sets_fine)):
            for j in range(i + 1, len(real_sets_fine)):
                out(f"REAL overlap fine:   Jaccard(Top-{args.topK})
batch{i+1} vs batch{j+1} = {jaccard(real_sets_fine[i],
real_sets_fine[j]):.3f}")
                out(f"REAL overlap coarse: Jaccard(Top-{args.topK})
batch{i+1} vs batch{j+1} = {jaccard(real_sets_coarse[i],
real_sets_coarse[j]):.3f}")

        out("")
        out("=== Notes (scientific reading) ===")
        out("1) Compare REAL vs NULL primarily via deltas/effect sizes
and batch stability, not raw entropy levels (d differs with n_qubits).")
        out("2) Use fine families for within-model discovery; use coarse
families for cross-model interpretability.")
        out("3) If results at n=4 resemble n=3 (stable deltas + stable
overlaps), that is strong qualitative evidence the effect is not a 3-
qubit artifact.")
        out("")
        out("=== End of v22 ===")


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