Topological Fault Tolerance, Fiedler Routability, and Partially Bridged Percolation

Finite Graph and Directional-Compatibility Models

Verification Code Companion

Charles D. Schaper, Ph.D.
Annals of the Chrysene Formalism, Volume 1, Number 4
Article 94, November 2026

1 Scope and reproducible result

The replacement program executes **29,784 explicit checks per mode**, including eight numerical checks (six spectral and two inherited floating-point comparisons). Normal Python and optimized Python produce byte-identical output. Ten deliberately false propositions are rejected. The program does not depend on Python `assert` statements, so optimized execution does not disable verification.

This is finite verification of the stated combinatorial and geometric models. It does not establish a synthetic route, independent chemical bond yields, biological inheritance, integral homology for arbitrary complexes or universal analytic graph theorems. Numerical Laplacian checks use NumPy and tolerance 10^{-10}; integer graph tests and rational probability/ceiling tests are exact. A check count counts calls to the explicit validation function, not individual Boolean clauses or distinct mathematical theorems.

2 Corrections to the supplied source

The supplied script fails its scalar-sum assertion: exhaustive enumeration gives $60,454,593$, not $254,121,840$. Its geometric thresholds also disagree with its own formula: the corrected ceilings are $233, 239, 243, 259$ for box heights $6, 10, 12, 18$. The source's historical all-pass statement is therefore superseded by this dossier. The original files remain provenance, not current verification evidence.

3 Coverage and interpretation

Directional counts.
All 4096 lateral bridge subsets are evaluated. An independent support-inclusion implementation agrees at every mask with the directional-count implementation; scalar degree counts are independently recomputed. Fixed-family, orientation-flexible projected-family, and orientation-resolved counts are kept distinct. The respective directional layer totals are $4,410,944$, $10,648,000$, and $64,000,000$. The flexible convention counts a family once if an orientation exists, rather than counting every orientation.

Exact probabilities.
Rational arithmetic checks the 95% threshold at half-heights $3, 5, 6, 9$ for yields $1, 0.95, 0.90, 0.80$, and failure at the preceding admissible height when applicable. Finite range, yield and height monotonicity tests accompany the manuscript's general argument. Independence is a model assumption, not an empirical finding.

Geometric ceilings.
The double-thickness model is checked without floating square roots. Write $S = 2a_x^2 + h_{\text{span}}^2$. The condition is $q \geq 0$ and $q^2 \geq 64S$, where $q = c^2L - S/4 - 64$. Exact rational checks verify each reported integer and reject its predecessor. The underlying constants and containment interpretation remain stipulated model inputs.

Cells and boundaries.
Cells are generated from their coordinate vertices, with signed integer boundary matrices. Both consecutive boundary products vanish. Boundary ranks over $\mathbb{F}_2$ are $(26, 28, 8)$ and $(53, 64, 20)$ for the two boxes; star complexes give $(6, 0, 0)$ and $(13, 0, 0)$. These computations corroborate the model; integral homology in the manuscript uses contractibility, not an inference from mod-two ranks alone.

Scission and metric bounds.
Every single deletion is checked on both stars and both boxes. All 1431 and 6786 double deletions on the boxes preserve connectivity. An explicit three-edge corner cut disconnects each box. For every box edge deletion and every vertex pair, the shortest-path bound is checked; the removed edge's endpoints have a three-edge replacement. A pair of complete four-vertex graphs joined by a bridge shows why minimum degree three alone does not imply edge connectivity three.

Spectra and minima.
The full numerical Cartesian-product Laplacian spectra are compared on both boxes. The 19-site plaquette-star and its eight extra corner sites are explicitly distinguished. None of these counts proves a molecular size minimum without the stated geometric and model restrictions.

4 Negative controls

The ten rejected mutations exchange fixed and flexible saturated counts, equate incompatible partition sums, lower the success threshold height, lower the geometric ceiling, claim a tree bypass, erase graph cycles, assert double-cut disconnection in the tested box, or conflate the 19-site plaquette-star with its 27-site corner closure. They are wrong propositions checked against independently computed outputs; they are not additional positive theorems.

5 Reproduction and provenance

Run `python run_verification.py` with Python 3 and NumPy installed. The runner checks the pinned SHA-256 source hash, runs both modes in temporary directories, compares all output bytes, and writes `execution_audit.json`. The generated macro file is the publication source of the tested numerical values. The executable source is embedded in this PDF and reproduced below; the standalone LaTeX recreates that exact source.

6 Complete executable source

Article 94 source

```
#!/usr/bin/env python3
"""Article 94 finite verification. Python 3 with numpy. No assert-dependent
    checks.
```

```
All probabilities use Fraction; graph invariants use explicit constructed
    cells.
Physical synthesis, arbitrary networks, integer homology and real spectral
analysis are not established by finite tests alone."""
from fractions import Fraction
from pathlib import Path
import json
import numpy as np
checks = 0
numerical_checks = 0
def check(value, label, numerical=False):
 global checks, numerical_checks
 checks += 1
 numerical_checks += int(numerical)
 if not value: raise RuntimeError(label)
"""
verify_a94.py
Verification script for Article 94: "Topological Fault Tolerance, Fiedler
    Routability,
and Partially Bridged Percolation of the Interior Chrysene Lattice" -- Final
    Revised Edition
Annals of the Chrysene Formalism, Vol. 1, No. 5, pp. 3000ff. (2026).

Uses Python standard library only (collections.deque, itertools, math).
Verifies Blocks S1--S6 and writes verification_values_a94.tex.
"""
from collections import deque
import itertools
import math
macros = {}

def emit(name, val):
    check(name not in macros, 'inherited predicate line 18')
    macros[name] = str(val)
X_EDGES = [((kx, ky), (kx + 1, ky)) for ky in range(3) for kx in range(2)]
Y_EDGES = [((kx, ky), (kx, ky + 1)) for kx in range(3) for ky in range(2)]
LAT_EDGES = X_EDGES + Y_EDGES
check(len(LAT_EDGES) == 12, 'inherited predicate line 27')

def count_layer_families(active_edges):
    dx_minus = {(kx, ky): 0 for kx in range(3) for ky in range(3)}
    dx_plus = {(kx, ky): 0 for kx in range(3) for ky in range(3)}
    deg_tot = {(kx, ky): 0 for kx in range(3) for ky in range(3)}
    for u, v in active_edges:
        deg_tot[u] += 1
        deg_tot[v] += 1
        if u[1] == v[1]:
            dx_plus[u] = 1
            dx_minus[v] = 1
    prod_scalar, prod_fixed, prod_orbit = (1, 1, 1)
    for pt in deg_tot:
        d = deg_tot[pt]
        n_scal = 3 if d <= 2 else 2 if d == 3 else 1
        n_fix = 1 + (1 if dx_plus[pt] == 0 else 0) + (1 if dx_plus[pt] == 0
            and dx_minus[pt] == 0 else 0)
        dx_sum = dx_plus[pt] + dx_minus[pt]
        n_orb = 1 + (1 if dx_sum <= 1 else 0) + (1 if dx_sum == 0 else 0)
        prod_scalar *= n_scal
        prod_fixed *= n_fix
```

```
        prod_orbit *= n_orb
    return (prod_scalar, prod_fixed, prod_orbit)
sat_scal, sat_fix, sat_orb = count_layer_families(LAT_EDGES)
check(sat_scal == 1296, 'inherited predicate line 56')
check(sat_scal * 4 ** 9 == 339738624, 'inherited predicate line 57')
check(sat_fix == 8, 'inherited predicate line 58')
check(sat_orb == 64, 'inherited predicate line 59')
sum_scal, sum_fix, sum_orb = (0, 0, 0)
for mask in range(1 << 12):
    sub = [LAT_EDGES[i] for i in range(12) if mask >> i & 1]
    ps, pf, po = count_layer_families(sub)
    sum_scal += ps
    sum_fix += pf
    sum_orb += po
check(sum_fix == 2 ** 6 * 41 ** 3 == 4410944, 'inherited predicate line 70')
check(sum_orb == 2 ** 6 * 55 ** 3 == 10648000, 'inherited predicate line 71')
check(sum_scal == 60454593, 'inherited predicate line 72')
emit('valScalarLayerFam', f'{sat_scal:,}')
emit('valScalarLayerFull', f'{sat_scal * 4 ** 9:,}')
emit('valDirFixedLayerFam', sat_fix)
emit('valDirOrbitLayerFam', sat_orb)
emit('valRowSumFixed', 41)
emit('valRowSumOrbit', 55)
emit('valPartSumFixed', f'{sum_fix:,}')
emit('valPartSumOrbit', f'{sum_orb:,}')
emit('valPartSumScalar', f'{sum_scal:,}')
emit('valBridgeSubsetsLayer', f'{1 << 12:,}')

def h_min_perc(pb, eps=0.05):
    p4 = pb ** 4
    h = 3
    while (1.0 - (1.0 - p4) ** (h - 2)) ** 2 < 1.0 - eps:
        h += 1
    return h

def l_crit_for_nz(nz):
    ax = 7.9
    h_span = 1.38 + 0.34 * (nz - 3)
    d_bl = 4.0
    c_rad = 0.89923
    r_circ = 0.5 * math.sqrt(2.0 * ax ** 2 + h_span ** 2) + d_bl
    return math.ceil(((r_circ + d_bl) / c_rad) ** 2)
h95 = h_min_perc(0.95)
h90 = h_min_perc(0.9)
h80 = h_min_perc(0.8)
check((h95, h90, h80) == (5, 6, 9), 'inherited predicate line 106', True)
L100 = l_crit_for_nz(6)
L95 = l_crit_for_nz(2 * h95)
L90 = l_crit_for_nz(2 * h90)
L80 = l_crit_for_nz(2 * h80)
check((L100, L95, L90, L80) == (233, 239, 243, 259), 'inherited predicate line
    112', True)
emit('valPcplNinetyFive', f'{0.95 ** 4:.4f}')
emit('valPcplNinety', f'{0.9 ** 4:.4f}')
emit('valPcplEighty', f'{0.8 ** 4:.4f}')
emit('valPheritBaseNinetyFive', f'{0.95 ** 8:.4f}')
emit('valPheritBaseNinety', f'{0.9 ** 8:.4f}')
emit('valHminNinetyFive', h95)
emit('valNodesMinNinetyFive', 18 * h95)
```

```
emit('valHminNinety', h90)
emit('valNodesMinNinety', 18 * h90)
emit('valHminEighty', h80)
emit('valNodesMinEighty', 18 * h80)
emit('valLcritOneHundred', L100)
emit('valLcritNinetyFive', L95)
emit('valLcritNinety', L90)
emit('valLcritEighty', L80)

def build_grid_graph(pts_set):
    pts = sorted(pts_set)
    edges = []
    for i, u in enumerate(pts):
        for v in pts[i + 1:]:
            if abs(u[0] - v[0]) + abs(u[1] - v[1]) + abs(u[2] - v[2]) == 1:
                edges.append((u, v))
    edge_set = set(edges) | set(((v, u) for u, v in edges))
    plaq = 0
    for u in pts:
        for d1, d2 in (((1, 0, 0), (0, 1, 0)), ((1, 0, 0), (0, 0, 1)), ((0, 1,
             0), (0, 0, 1))):
            p1 = (u[0] + d1[0], u[1] + d1[1], u[2] + d1[2])
            p2 = (u[0] + d2[0], u[1] + d2[1], u[2] + d2[2])
            p3 = (u[0] + d1[0] + d2[0], u[1] + d1[1] + d2[1], u[2] + d1[2] +
                d2[2])
            if (u, p1) in edge_set and (u, p2) in edge_set and ((p1, p3) in
                edge_set) and ((p2, p3) in edge_set):
                plaq += 1
    cubes = 0
    for x, y, z in pts:
        if all(((x + dx, y + dy, z + dz) in pts_set for dx in (0, 1) for dy in
             (0, 1) for dz in (0, 1))):
            cubes += 1
    return (pts, edges, plaq, cubes)

def is_connected(pts, active_edges):
    adj = {u: [] for u in pts}
    for u, v in active_edges:
        adj[u].append(v)
        adj[v].append(u)
    vis = {pts[0]}
    q = deque([pts[0]])
    while q:
        curr = q.popleft()
        for nxt in adj[curr]:
            if nxt not in vis:
                vis.add(nxt)
                q.append(nxt)
    return len(vis) == len(pts)
X7_pts = {(0, 0, 0), (1, 0, 0), (-1, 0, 0), (0, 1, 0), (0, -1, 0), (0, 0, 1),
    (0, 0, -1)}
X14_pts = X7_pts | {(x, y, z + 3) for x, y, z in X7_pts}
B333_pts = {(x, y, z) for x in range(3) for y in range(3) for z in range(3)}
B336_pts = {(x, y, z) for x in range(3) for y in range(3) for z in range(6)}
_, X7_e, X7_p, X7_c = build_grid_graph(X7_pts)
_, X14_e, X14_p, X14_c = build_grid_graph(X14_pts)
_, B333_e, B333_p, B333_c = build_grid_graph(B333_pts)
_, B336_e, B336_p, B336_c = build_grid_graph(B336_pts)
check((len(X7_pts), len(X7_e), len(X7_e) - len(X7_pts) + 1, X7_p, X7_c) == (7,
```

```
    6, 0, 0, 0), 'inherited predicate line 180')
check((len(X14_pts), len(X14_e), len(X14_e) - len(X14_pts) + 1, X14_p, X14_c)
    == (14, 13, 0, 0, 0), 'inherited predicate line 181')
check((len(B333_pts), len(B333_e), len(B333_e) - len(B333_pts) + 1, B333_p,
    B333_c) == (27, 54, 28, 36, 8), 'inherited predicate line 182')
check((len(B336_pts), len(B336_e), len(B336_e) - len(B336_pts) + 1, B336_p,
    B336_c) == (54, 117, 64, 84, 20), 'inherited predicate line 183')
check(B333_p - B333_c == 28, 'inherited predicate line 184')
check(B336_p - B336_c == 64, 'inherited predicate line 185')
check(all((not is_connected(list(X7_pts), X7_e[:i] + X7_e[i + 1:]) for i in
    range(len(X7_e)))), 'inherited predicate line 188')
check(all((not is_connected(list(X14_pts), X14_e[:i] + X14_e[i + 1:]) for i in
    range(len(X14_e)))), 'inherited predicate line 189')
overlap_X7 = max((len(X7_pts & {(x + dx, y + dy, z + dz) for x, y, z in X7_pts
    }) for dx in (-2, -1, 0, 1, 2) for dy in (-2, -1, 0, 1, 2) for dz in (-2,
    -1, 0, 1, 2) if (dx, dy, dz) != (0, 0, 0)))
overlap_B333 = len(B333_pts & {(x + 1, y, z) for x, y, z in B333_pts})
check((overlap_X7, overlap_B333) == (2, 18), 'inherited predicate line 198')
b333_list = list(B333_pts)
b336_list = list(B336_pts)
pairs_333 = sum((1 for i, j in itertools.combinations(range(len(B333_e)), 2)
    if is_connected(b333_list, [B333_e[k] for k in range(len(B333_e)) if k !=
    i and k != j])))
pairs_336 = sum((1 for i, j in itertools.combinations(range(len(B336_e)), 2)
    if is_connected(b336_list, [B336_e[k] for k in range(len(B336_e)) if k !=
    i and k != j])))
check((pairs_333, pairs_336) == (1431, 6786), 'inherited predicate line 210')
fiedler_333 = 2.0 - 2.0 * math.cos(math.pi / 3.0)
fiedler_336 = 2.0 - 2.0 * math.cos(math.pi / 6.0)
emit('valCrossSevenNodes', 7)
emit('valCrossSevenEdges', 6)
emit('valCrossSevenBetti', 0)
emit('valCrossSevenPlaq', 0)
emit('valCrossFourteenNodes', 14)
emit('valCrossFourteenEdges', 13)
emit('valCrossFourteenBetti', 0)
emit('valCrossFourteenPlaq', 0)
emit('valBoxThreeNodes', 27)
emit('valBoxThreeEdges', 54)
emit('valBoxThreeBetti', 28)
emit('valBoxThreePlaq', 36)
emit('valBoxThreeCubes', 8)
emit('valBoxThreeFiedler', f'{fiedler_333:.4f}')
emit('valBoxSixNodes', 54)
emit('valBoxSixEdges', 117)
emit('valBoxSixBetti', 64)
emit('valBoxSixPlaq', 84)
emit('valBoxSixCubes', 20)
emit('valBoxSixFiedler', f'{fiedler_336:.4f}')
emit('valBoxEdgeConn', 3)
emit('valBoxThreeDoublePairs', f'{pairs_333:,}')
emit('valBoxSixDoublePairs', f'{pairs_336:,}')
emit('valCrossSevenOverlap', overlap_X7)
emit('valBoxThreeOverlap', overlap_B333)
emit('valCrossSevered', 1)
emit('valBoxSevered', 9)
# Independent support-subset comparison for every bridge mask.
# A family is counted once if SOME orientation is feasible; orientations are
# not counted as additional microstates in the orbit-family convention.
```

```
D={2,3}; I={0,2,3}; Irev={1,2,3}; T={0,1,2,3}
oriented_sum=0
scalar_independent=0
for mask in range(4096):
    active=[LAT_EDGES[i] for i in range(12) if mask>>i&1]
    need={p:set() for p in itertools.product(range(3),repeat=2)}
    for u,v in active:
        a,b=(1,0) if u[1]==v[1] else (3,2)
        need[u].add(a); need[v].add(b)
    fixed=math.prod(sum(s<=f for f in (D,I,T)) for s in need.values())
    orbit=math.prod(int(s<=D)+int(s<=I or s<=Irev)+int(s<=T) for s in need.
        values())
    ps,pf,po=count_layer_families(active)
    scalar=math.prod(sum(len(t)>=len(s) for t in (D,I,T)) for s in need.values
        ())
    scalar_independent+=scalar
    oriented_sum+=math.prod(sum(s<=f for f in (D,I,Irev,T)) for s in need.
        values())
    check(scalar==ps,"scalar independent degree convention")
    check(fixed==pf,'support-subset fixed count')
    check(orbit==po,'existential-orientation family count')

check(oriented_sum==64000000,"orientation resolved layer")
check(scalar_independent==60454593,"scalar independent layer sum")

# Exact success probabilities, minimal half-heights and monotonicity.
def probability(p,h):
    return (1-(1-p**4)**(h-2))**2
for p,expected in [(Fraction(1),3),(Fraction(19,20),5),(Fraction(9,10),6),(
    Fraction(4,5),9)]:
    check(probability(p,expected)>=Fraction(19,20),'threshold attained')
    check(expected==3 or probability(p,expected-1)<Fraction(19,20),'threshold
        minimal')
for k in range(1,21):
    p=Fraction(k,20)
    for h in range(3,20):
        check(0<=probability(p,h)<=1,'probability range')
        check(probability(p,h+1)>=probability(p,h),'height monotonicity')
        if k<20: check(probability(p+Fraction(1,20),h)>=probability(p,h),'
            yield monotonicity')

# Exact rational ceiling certification of the stated geometric model.
# (sqrt(S)/2+8)^2 = S/4+64+8sqrt(S).  All constants are decimal rationals.
def lumen_fits(L,nz):
    S=2*Fraction(790,100)**2+(Fraction(138,100)+Fraction(34,100)*(nz-3))**2
    q=Fraction(89923,100000)**2*L-S/4-64
    return q>=0 and q*q>=64*S
for nz,L in [(6,233),(10,239),(12,243),(18,259)]:
    check(lumen_fits(L,nz),'geometric ceiling sufficient')
    check(not lumen_fits(L-1,nz),'geometric ceiling minimal')

def gf2rank(columns):
    piv={}
    for c in columns:
        while c:
            i=c.bit_length()-1
            if i in piv: c^=piv[i]
            else: piv[i]=c; break
    return len(piv)
```

```
def cells(points):
    points=set(points); out=[]
    for d in range(4):
        out.append([(p,axes) for p in sorted(points)
                    for axes in itertools.combinations(range(3),d)
                    if all(tuple(p[i]+int(i in axes and bits[axes.index(i)])
                        for i in range(3)) in points
                            for bits in itertools.product((0,1),repeat=d))])
    return out

def boundaries(cs):
    matrices=[]
    for d in range(1,4):
        index={c:i for i,c in enumerate(cs[d-1])}
        cols=[]
        for p,axes in cs[d]:
            col=[0]*len(index)
            for j,a in enumerate(axes):
                rest=axes[:j]+axes[j+1:]
                q=tuple(p[i]+int(i==a) for i in range(3))
                col[index[(q,rest)]]+=(-1)**j
                col[index[(p,rest)]]-=(-1)**j
            cols.append(col)
        matrices.append(np.array(cols,dtype=np.int64).reshape((len(cols),len(
            index))).T)
    return matrices

def distances(pts,edges,start):
    adj={v:[] for v in pts}
    for u,v in edges: adj[u].append(v);adj[v].append(u)
    d={start:0}; q=deque([start])
    while q:
        u=q.popleft()
        for v in adj[u]:
            if v not in d: d[v]=d[u]+1;q.append(v)
    return d

model_results={}
for name,points,expected in [('X7',X7_pts,(7,6,0,0)),('X14',X14_pts
    ,(14,13,0,0)),
                             ('B333',B333_pts,(27,54,36,8)),('B336',B336_pts
                                 ,(54,117,84,20))]:
    cs=cells(points); b=boundaries(cs)
    check(tuple(map(len,cs))==expected,'explicit cubical cells '+name)
    ranks=[gf2rank([sum((int(x)%2)<<i for i,x in enumerate(col)) for col in
        mat.T]) for mat in b]
    check(ranks[0]==len(points)-1,'boundary one rank '+name)
    check(ranks[1]==len(cs[1])-len(points)+1,'vanishing H1 over GF2 '+name)
    check(ranks[2]==len(cs[3]),'boundary three injective '+name)
    for j in (0,1): check(np.count_nonzero(b[j]@b[j+1])==0,'boundary squared
        zero '+name)
    pts,edges,_,_=build_grid_graph(points)
    lap=np.zeros((len(pts),len(pts)))
    ix={p:i for i,p in enumerate(pts)}
    for u,v in edges:
        i,j=ix[u],ix[v];lap[i,i]+=1;lap[j,j]+=1;lap[i,j]-=1;lap[j,i]-=1
    eig=np.linalg.eigvalsh(lap)
    check(abs(eig[0])<1e-10 and eig[1]>1e-9,'numerical Laplacian connectivity
```

```
        '+name,True)
    if name.startswith('B'):
        nz=3 if name=='B333' else 6
        predicted=sorted(sum(2-2*math.cos(math.pi*k/n) for k,n in zip(ks,(3,3,
            nz)))
                         for ks in itertools.product(range(3),range(3),range(
                             nz)))
        check(max(abs(x-y) for x,y in zip(eig,predicted))<1e-10,'full
            Cartesian spectrum '+name,True)
    for i,e in enumerate(edges):
        remaining=edges[:i]+edges[i+1:]
        connected=is_connected(pts,remaining)
        check(connected==name.startswith('B'),'single-edge sweep '+name)
        if name.startswith('B'):
            check(distances(pts,remaining,e[0])[e[1]]==3,'minimal local bypass
                '+name)
            for u in pts:
                old=distances(pts,edges,u); new=distances(pts,remaining,u)
                check(all(new[v]<=old[v]+2<=3*old[v] for v in pts if v!=u),'
                    global bypass bound '+name)
    if name.startswith('B'):
        for i,j in itertools.combinations(range(len(edges)),2):
            check(is_connected(pts,[e for k,e in enumerate(edges) if k not in
                (i,j)]),'double-edge sweep '+name)
        corner=min(points); cut=[e for e in edges if corner in e]
        check(len(cut)==3,'three-edge upper bound '+name)
        check(not is_connected(pts,[e for e in edges if corner not in e]),'
            corner disconnection '+name)
    model_results[name]={'cells':list(expected),'boundary_ranks_GF2':ranks,'
        lambda2':float(eig[1])}

# A plaquette-complete center needs 19 sites, before corner closure.
P19={p for p in itertools.product((-1,0,1),repeat=3) if sum(c!=0 for c in p)
    <=2}
check(len(P19)==19,'plaquette-star minimum')
check(len(set(itertools.product((-1,0,1),repeat=3))-P19)==8,'extra corners')
check(all(tuple(s if i==a else t if i==b else 0 for i in range(3)) in P19
          for a,b in itertools.combinations(range(3),2) for s,t in itertools.
              product((-1,1),repeat=2)),
      'all twelve central plaquettes')
# Minimum degree alone does not establish edge connectivity: two K4 linked
    once.
k4edges=[(i,j) for block in (range(4),range(4,8)) for i,j in itertools.
    combinations(block,2)]+[(0,4)]
check(min(sum(v in e for e in k4edges) for v in range(8))==3,'min-degree
    counterexample')
check(not is_connected(list(range(8)),k4edges[:-1]),'bridge counterexample')

# Ten deliberately wrong propositions must be rejected by substantive outputs.
mutations=[('saturated fixed',sat_fix==64),('saturated orbit',sat_orb==8),
 ('partial fixed',sum_fix==sum_orb),('partial scalar',sum_scal==sum_fix),
 ('threshold95',probability(Fraction(19,20),4)>=Fraction(19,20)),
 ('lumen232',lumen_fits(232,6)),('tree bypass',is_connected(sorted(X7_pts),
     X7_e[1:])),
 ('zero graph cycles',len(B333_e)-len(B333_pts)+1==0),
 ('two cuts disconnect',pairs_336!=math.comb(117,2)),('plaquette closure27',
     len(P19)==27)]
for label,bad in mutations: check(not bad,'negative control '+label)
report={'article':94,'checks':checks,'exact_checks':checks-numerical_checks,'
```

```
    numerical_checks':numerical_checks,
366         'negative_controls':len(mutations),'models':model_results,
        'scope':'Finite tests; GF2 homology ranks do not alone prove integral
           homology. Spectra numerical; physical realization excluded.'}
Path('verification_report.json').write_text(json.dumps(report,indent=2)+'\n')
Path('verification_values_a94.tex').write_text('% Generated by the checked
    Article 94 program.\n'+''.join('\\newcommand{\\'+k+'}{'+v+'}\n' for k,v in
     macros.items()))
print(json.dumps(report,sort_keys=True))
```