ANNALS OF THE CHRYSENE FORMALISM
COMPUTATIONAL VERIFICATION DOSSIER | ARTICLE 102 | VOL. 1, NO. 5 | NOVEMBER 2026

Verification Code: Article 102

Arithmetic Geometry, Spectral Zeta Theory, and Non-Commutative Geometry of the Chrysene Formalism

Charles D. Schaper, Ph.D.

Executed finite-model verification. The complete Python program below was run successfully during preparation of this publication package. It uses only the Python standard library and emits 43 manuscript macros. Its graph claims refer to the two explicit adjacency models in the listing. Execution is independent of LaTeX compilation.

1 Scope and corrections

The script exhaustively checks finite-field operations, the eight affine and nine projective points, the finite elliptic-curve group law, both evaluation-code word sets and weight distributions, graph matching counts, Laplacian cofactors, determinant-polynomial coefficients, Hashimoto traces, inversion/grading identities, and rational row/column-sum certificates for path-distance lower bounds.

The independent audit corrects four quantities in the supplied draft: the length-12 Hashimoto trace is **120**, not 192; the 3-matching counts are **816 and 814**, not 810 and 808; the sixth adjacency characteristic coefficients are -824 **and** -822, not -818 and -816; and $P'_G(1) = \mathbf{7134}$, not -7134. Inversion swaps the bipartite halves, so $J\gamma = -\gamma J$.

The code does not prove the general Riemann–Roch or Hasse–Weil theorems, a universal catacondensed-tree recurrence, a full real spectral triple with the order-one condition, a physical molecular realization, or chemical performance. The genus and tensor-product interpretation are mathematical inputs discussed in the manuscript, not discoveries made by the program. The extension-point formula is evaluated at four degrees, not derived computationally over all extension fields. The real path-distance lower bound follows analytically from the norm inequality; the program verifies its finite rational certificates.

2 Independent checks and reproducibility

The original code inferred the 3-matching count from an erroneous formula. The revised code additionally counts matchings directly by deletion recursion, obtains the adjacency characteristic coefficients by Newton identities, and obtains the Ihara–Bass polynomial by dividing the Hashimoto determinant polynomial by $(1-u^2)^3$. Exact determinant evaluations at $u = 1/2$ and $u = 2$ cross-check this polynomial. The derivative at $u = 1$ is then checked from its coefficients. No floating-point comparison is needed for these claims.

Save the listing as `verify_a102.py`, then run `python3 verify_a102.py`. Use the supplied raw source rather than PDF text extraction when possible. The output file `verification_values_a102.tex` is optional for the standalone manuscript, whose fallback values agree with this run.

Companion Artifact: Schaper, C. D. (2026). Arithmetic Geometry, Spectral Zeta Theory, and Non-Commutative Geometry of the Chrysene Formalism: The (9, 8) Hermitian Curve over GF(4), Genus-1 Goppa Codes, the Ihara–Bass Zeta Function, and Connes' Spectral Triple. *Annals of the Chrysene Formalism*, 1(5), 1966–1974 https://chrysene.com/index.php/acf/article/view/102 • © 2026 Charles D. Schaper.

3 Complete executable source

```python
#!/usr/bin/env python3
"""
verify_a102.py
Verification script for Article 102: "Arithmetic Geometry, Spectral Zeta Theory,
and Non-Commutative Geometry of the Chrysene Formalism"
Annals of the Chrysene Formalism, Vol. 1, No. 5, pp. 1966--1974 (2026).

Uses Python standard library only (fractions, itertools).
Verifies Blocks Z1--Z7 and writes verification_values_a102.tex.
"""

from fractions import Fraction
import itertools

macros = {}
def emit(name, val):
    assert name not in macros, f"Duplicate macro: {name}"
    macros[name] = str(val)

# ======================================================================
# BLOCK Z1: Hermitian Curve E_H: y^2 + y = x^3 over GF(4)
# ======================================================================
GF4 = (0, 1, 2, 3)
def gf4_add(a, b):
    return a ^ b

MUL_TABLE = (
    (0, 0, 0, 0),
    (0, 1, 2, 3),
    (0, 2, 3, 1),
    (0, 3, 1, 2),
)
def gf4_mul(a, b):
    return MUL_TABLE[a][b]

def gf4_pow(a, e):
    res = 1
    for _ in range(e):
        res = gf4_mul(res, a)
    return res

def gf4_dot(u, v):
    acc = 0
    for a, b in zip(u, v):
        acc = gf4_add(acc, gf4_mul(a, b))
    return acc

def wt(vec):
    return sum(1 for x in vec if x != 0)

# Enumerate affine rational points of y^2 + y = x^3 in GF(4)^2
D_aff = []
for x in GF4:
    for y in GF4:
        lhs = gf4_add(gf4_pow(y, 2), y)
        rhs = gf4_pow(x, 3)
```

```
        if lhs == rhs:
            D_aff.append((x, y))

assert len(D_aff) == 8
tot_pts = len(D_aff) + 1  # + P_infty = [0:1:0]
assert tot_pts == 9 == 4 + 1 + 2 * 1 * 2

# Verify every affine point is a flex point (tangent line intersects with multiplicity 3)
for x0, y0 in D_aff:
    slope = gf4_pow(x0, 2)
    intersections = [x for x in GF4 if gf4_add(gf4_pow(gf4_add(y0, gf4_mul(slope, gf4_add(x, x0))), 2),
                     gf4_add(gf4_add(y0, gf4_mul(slope, gf4_add(x, x0))), gf4_pow(x, 3))) == 0]
    assert intersections == [x0]

emit("valHermGenus", 1)
emit("valHermTotPts", tot_pts)
emit("valHermAffPts", len(D_aff))

# ============================================================================
# BLOCK Z2: Genus-1 Hermitian Goppa Codes C_{Herm,4} and C_{Herm,3}
# ============================================================================
G_Herm4 = [
    tuple(1 for (x, y) in D_aff),
    tuple(x for (x, y) in D_aff),
    tuple(y for (x, y) in D_aff),
    tuple(gf4_pow(x, 2) for (x, y) in D_aff),
]

# Verify the Euclidean generator Gram matrix is zero (self-orthogonality).
for r0 in range(4):
    for r1 in range(4):
        assert gf4_dot(G_Herm4[r0], G_Herm4[r1]) == 0

C_Herm4 = []
for c0, c1, c2, c3 in itertools.product(GF4, repeat=4):
    coeffs = (c0, c1, c2, c3)
    cw = tuple(
        gf4_add(
            gf4_add(gf4_mul(coeffs[0], G_Herm4[0][j]), gf4_mul(coeffs[1], G_Herm4[1][j])),
            gf4_add(gf4_mul(coeffs[2], G_Herm4[2][j]), gf4_mul(coeffs[3], G_Herm4[3][j]))
        )
        for j in range(8)
    )
    C_Herm4.append(cw)

assert len(set(C_Herm4)) == 256
wt_dist_4 = [sum(1 for cw in C_Herm4 if wt(cw) == w) for w in range(9)]
assert wt_dist_4 == [1, 0, 0, 0, 18, 96, 24, 96, 21]
d_min_4 = min(wt(cw) for cw in C_Herm4 if wt(cw) > 0)
assert d_min_4 == 4

C_Herm3 = []
for c0, c1, c2 in itertools.product(GF4, repeat=3):
    cw = tuple(
        gf4_add(gf4_add(gf4_mul(c0, G_Herm4[0][j]), gf4_mul(c1, G_Herm4[1][j])), gf4_mul(c2, G_Herm4[2][j]))
        for j in range(8)
    )
```

```
    C_Herm3.append(cw)

assert len(set(C_Herm3)) == 64
wt_dist_3 = [sum(1 for cw in C_Herm3 if wt(cw) == w) for w in range(9)]
assert wt_dist_3 == [1, 0, 0, 0, 0, 24, 12, 24, 3]
d_min_3 = min(wt(cw) for cw in C_Herm3 if wt(cw) > 0)
assert d_min_3 == 5

emit("valHermFourDim", 4)
emit("valHermFourDist", d_min_4)
emit("valHermFourWords", len(C_Herm4))
emit("valHermFourWtFour", wt_dist_4[4])
emit("valHermFourWtFive", wt_dist_4[5])
emit("valHermFourWtSix", wt_dist_4[6])
emit("valHermFourWtSeven", wt_dist_4[7])
emit("valHermFourWtEight", wt_dist_4[8])

emit("valHermThreeDim", 3)
emit("valHermThreeDist", d_min_3)
emit("valHermThreeWords", len(C_Herm3))
emit("valHermThreeWtFive", wt_dist_3[5])
emit("valHermThreeWtSix", wt_dist_3[6])
emit("valHermThreeWtSeven", wt_dist_3[7])
emit("valHermThreeWtEight", wt_dist_3[8])

# ======================================================================
# BLOCK Z3 & Z4: Spanning-Tree Complexity & Ihara-Bass / Zagreb Split
# ======================================================================
V_A_chr = ["v1", "v3", "v5", "v8", "v10", "v12", "w2", "w3", "w6"]
V_B_chr = ["v2", "v4", "v6", "v7", "v9", "v11", "w1", "w4", "w5"]
V_CHR = V_A_chr + V_B_chr
IDX_CHR = {v: i for i, v in enumerate(V_CHR)}
ADJ_CHR = {
    "v1": ["v2", "w1"], "v2": ["v1", "v3"], "v3": ["v2", "v4"], "v4": ["v3", "w3"],
    "v5": ["w5", "v6"], "v6": ["v5", "w6"], "v7": ["w6", "v8"], "v8": ["v7", "v9"],
    "v9": ["v8", "v10"], "v10": ["v9", "w4"], "v11": ["w2", "v12"], "v12": ["v11", "w1"],
    "w1": ["v12", "v1", "w3"], "w2": ["v11", "w4", "w5"], "w3": ["v4", "w1", "w5"],
    "w4": ["v10", "w2", "w6"], "w5": ["v5", "w2", "w3"], "w6": ["v6", "v7", "w4"],
}
A_chr = [[0]*18 for _ in range(18)]
for u, nbrs in ADJ_CHR.items():
    for v in nbrs:
        A_chr[IDX_CHR[u]][IDX_CHR[v]] = 1

V_A_tet = [f"u{2*k}" for k in range(5)] + [f"v{2*k+1}" for k in range(4)]
V_B_tet = [f"v{2*k}" for k in range(5)] + [f"u{2*k+1}" for k in range(4)]
V_TET = V_A_tet + V_B_tet
IDX_TET = {v: i for i, v in enumerate(V_TET)}
A_tet = [[0]*18 for _ in range(18)]
for k in range(8):
    u_a, u_b = IDX_TET[f"u{k}"], IDX_TET[f"u{k+1}"]
    v_a, v_b = IDX_TET[f"v{k}"], IDX_TET[f"v{k+1}"]
    A_tet[u_a][u_b] = A_tet[u_b][u_a] = 1
    A_tet[v_a][v_b] = A_tet[v_b][v_a] = 1
for k in (0, 2, 4, 6, 8):
    u_a, v_a = IDX_TET[f"u{k}"], IDX_TET[f"v{k}"]
    A_tet[u_a][v_a] = A_tet[v_a][u_a] = 1

```

```
def exact_det(M):
    n = len(M)
    A = [[Fraction(M[r][c]) for c in range(n)] for r in range(n)]
    sign = 1
    for i in range(n):
        pivot = i
        while pivot < n and A[pivot][i] == 0:
            pivot += 1
        if pivot == n:
            return Fraction(0)
        if pivot != i:
            A[i], A[pivot] = A[pivot], A[i]
            sign = -sign
        for r in range(i + 1, n):
            factor = A[r][i] / A[i][i]
            for c in range(i, n):
                A[r][c] -= factor * A[i][c]
    prod = Fraction(sign)
    for i in range(n):
        prod *= A[i][i]
    return prod

def spanning_trees(A_mat):
    n = len(A_mat)
    deg = [sum(A_mat[i]) for i in range(n)]
    L_minor = [[(deg[i] if i == j else -A_mat[i][j]) for j in range(n - 1)] for i in range(n - 1)]
    return int(exact_det(L_minor))

kappa_chr = spanning_trees(A_chr)
kappa_tet = spanning_trees(A_tet)
assert kappa_chr == kappa_tet == 1189
ihara_deriv = 2 * (21 - 18) * kappa_chr
assert ihara_deriv == 7134

def zagreb_Q(A_mat):
    n = len(A_mat)
    q_diag = [sum(A_mat[i]) - 1 for i in range(n)]
    total = 0
    for i in range(n):
        for j in range(i + 1, n):
            if A_mat[i][j] == 1:
                total += q_diag[i] * q_diag[j]
    return total

zq_chr = zagreb_Q(A_chr)
zq_tet = zagreb_Q(A_tet)
assert (zq_chr, zq_tet) == (44, 42)
m3_chr, m3_tet = 772 + zq_chr, 772 + zq_tet
a6_chr, a6_tet = -(m3_chr + 8), -(m3_tet + 8)
assert (m3_chr, m3_tet, a6_chr, a6_tet) == (816, 814, -824, -822)

emit("valCycleRank", 4)
emit("valDetQ", 64)
emit("valSpanTrees", f"{kappa_chr:,}")
emit("valIharaDerivOne", f"{ihara_deriv:,}")
emit("valZagrebQChr", zq_chr)
emit("valZagrebQTet", zq_tet)
emit("valMatchThreeChr", m3_chr)
```

```
emit("valMatchThreeTet", m3_tet)
emit("valSachsSixChr", a6_chr)
emit("valSachsSixTet", a6_tet)

# ======================================================================
# BLOCK Z5 & Z6: Even Spectral Data & Connes Spectral Distance Bounds
# ======================================================================
# Verify inversion commutes with A_chr; the grading sign is audited below.
IOTA_MAP = {
    "v1": "v7", "v7": "v1", "v2": "v8", "v8": "v2", "v3": "v9", "v9": "v3",
    "v4": "v10", "v10": "v4", "v5": "v11", "v11": "v5", "v6": "v12", "v12": "v6",
    "w1": "w6", "w6": "w1", "w2": "w5", "w5": "w2", "w3": "w4", "w4": "w3",
}
for u in V_CHR:
    for v in V_CHR:
        assert A_chr[IDX_CHR[u]][IDX_CHR[v]] == A_chr[IDX_CHR[IOTA_MAP[u]]][IDX_CHR[IOTA_MAP[v]]]

connes_6 = (6 - 1) / 2.0
connes_54 = (54 - 1) / 2.0
assert (connes_6, connes_54) == (2.50, 26.50)

emit("valDimQOne", 4)
emit("valDimQSix", "4,096")
emit("valConnesLowerSix", f"{connes_6:.2f}")
emit("valConnesLowerFiftyFour", f"{connes_54:.2f}")

# ======================================================================
# BLOCK Z7: Supersingular Extension Tower & Hashimoto B_G Walk Split
# ======================================================================
ext_pts = [(2**r - ((-1)**r))**2 for r in (1, 2, 3, 4)]
assert ext_pts == [9, 9, 81, 225]

def build_hashimoto(A_mat):
    n = len(A_mat)
    dir_edges = [(u, v) for u in range(n) for v in range(n) if A_mat[u][v] == 1]
    assert len(dir_edges) == 42
    B = [[0]*42 for _ in range(42)]
    for i, (u1, v1) in enumerate(dir_edges):
        for j, (u2, v2) in enumerate(dir_edges):
            if v1 == u2 and v2 != u1:
                B[i][j] = 1
    return B

def mat_mul_int(M1, M2):
    n = len(M1)
    return [[sum(M1[r][k] * M2[k][c] for k in range(n)) for c in range(n)] for r in range(n)]

def mat_trace(M):
    return sum(M[i][i] for i in range(len(M)))

B_chr = build_hashimoto(A_chr)
B_tet = build_hashimoto(A_tet)
B2_chr = mat_mul_int(B_chr, B_chr)
B2_tet = mat_mul_int(B_tet, B_tet)

P_chr_k = [[1 if r == c else 0 for c in range(42)] for r in range(42)]
P_tet_k = [[1 if r == c else 0 for c in range(42)] for r in range(42)]
traces_chr, traces_tet = {}, {}
```

```
for k in (2, 4, 6, 8, 10, 12, 14):
    P_chr_k = mat_mul_int(P_chr_k, B2_chr)
    P_tet_k = mat_mul_int(P_tet_k, B2_tet)
    traces_chr[k] = mat_trace(P_chr_k)
    traces_tet[k] = mat_trace(P_tet_k)

for k in (2, 4, 6, 8, 10, 12):
    assert traces_chr[k] == traces_tet[k]
assert (traces_chr[6], traces_chr[8], traces_chr[10], traces_chr[12]) == (48, 0, 60, 120)
assert (traces_chr[14], traces_tet[14]) == (168, 56)
assert traces_chr[14] // traces_tet[14] == 3

emit("valFrobTrace", -4)
emit("valHermExtTwo", ext_pts[1])
emit("valHermExtThree", ext_pts[2])
emit("valHermExtFour", ext_pts[3])
emit("valHashimotoDim", 42)
emit("valNbtSix", traces_chr[6])
emit("valNbtTen", traces_chr[10])
emit("valNbtTwelve", traces_chr[12])
emit("valNbtFourteenChr", traces_chr[14])
emit("valNbtFourteenTet", traces_tet[14])
emit("valNbtFourteenRatio", traces_chr[14] // traces_tet[14])

# Independent exact audit, added during publication preparation.
# These tests check the supplied finite models, not empirical realizations.
for a,b,c in itertools.product(GF4, repeat=3):
    assert gf4_add(gf4_add(a,b),c) == gf4_add(a,gf4_add(b,c))
    assert gf4_mul(gf4_mul(a,b),c) == gf4_mul(a,gf4_mul(b,c))
    assert gf4_mul(a,gf4_add(b,c)) == gf4_add(gf4_mul(a,b),gf4_mul(a,c))
for a in GF4:
    assert gf4_add(a,a)==0 and gf4_mul(a,1)==a
    if a: assert gf4_pow(a,3)==1

# Homogeneous smoothness and canonical representatives in P^2(GF4).
projective=[]
for X,Y,Z in itertools.product(GF4,repeat=3):
    if (X,Y,Z)==(0,0,0): continue
    first=next(t for t in (X,Y,Z) if t)
    if first!=1: continue
    if gf4_add(gf4_mul(gf4_pow(Y,2),Z),gf4_mul(Y,gf4_pow(Z,2)))==gf4_pow(X,3):
        projective.append((X,Y,Z))
        assert (gf4_pow(X,2),gf4_pow(Z,2),gf4_pow(Y,2))!=(0,0,0)
assert len(projective)==9
assert [p for p in projective if p[2]==0]==[(0,1,0)]

# Actual group law with O=None; associative on all 9^3 triples.
def point_add(P,Q):
    if P is None: return Q
    if Q is None: return P
    x,y=P; u,v=Q
    if x==u and y!=v: return None
    lam=gf4_pow(x,2) if P==Q else gf4_mul(gf4_add(y,v),gf4_pow(gf4_add(x,u),2))
    z=gf4_add(gf4_add(gf4_pow(lam,2),x),u)
    w=gf4_add(gf4_add(gf4_mul(lam,gf4_add(x,z)),y),1)
    return (z,w)
points=[None]+D_aff
for P,Q,R in itertools.product(points,repeat=3):
```

```
    assert point_add(P,Q) in points
    assert point_add(point_add(P,Q),R)==point_add(P,point_add(Q,R))
for P in points:
    assert point_add(point_add(P,P),P) is None

# Polynomial tangent identity: coefficient of t^2 is x0^4+x0=0,
# coefficient of t is x0^2+x0^2=0; residual is t^3.
for x0,y0 in D_aff:
    assert gf4_add(gf4_pow(x0,4),x0)==0
    assert gf4_add(gf4_pow(x0,2),gf4_pow(x0,2))==0

from functools import lru_cache

def matching_counts(A):
    n=len(A)
    @lru_cache(None)
    def rec(mask):
        if mask==0: return (1,)
        i=(mask & -mask).bit_length()-1
        rest=mask & ~(1<<i)
        out=list(rec(rest))+[0]
        for j in range(i+1,n):
            if rest & (1<<j) and A[i][j]:
                for k,v in enumerate(rec(rest & ~(1<<j))):
                    out[k+1]+=v
        while len(out)>1 and out[-1]==0: out.pop()
        return tuple(out)
    return rec((1<<n)-1)

# Newton identities compute determinant-polynomial coefficients exactly.
def det_identity_coefficients(M):
    n=len(M); power=[[int(i==j) for j in range(n)] for i in range(n)]
    tr=[0]; coeff=[1]
    for k in range(1,n+1):
        power=mat_mul_int(power,M); tr.append(mat_trace(power))
        numerator=-sum(coeff[k-i]*tr[i] for i in range(1,k+1))
        assert numerator % k==0
        coeff.append(numerator//k)
    return coeff

polynomials={}
for name,A,expect_m3,expect_pm,expect_sachs in [('chr',A_chr,816,8,-824),('tet',A_tet,814,5,-822)]:
    n=len(A); deg=[sum(row) for row in A]
    assert sum(deg)==42 and sorted(deg)==[2]*12+[3]*6
    assert all(A[i][j]==A[j][i] for i in range(n) for j in range(n))
    assert all(A[i][j]==0 for i in range(9) for j in range(9))
    assert all(A[i][j]==0 for i in range(9,18) for j in range(9,18))
    mc=matching_counts(A)
    assert mc[3]==expect_m3 and mc[9]==expect_pm
    char=det_identity_coefficients(A)
    assert char[6]==expect_sachs
    assert exact_det(A)==-expect_pm**2
    Q=[d-1 for d in deg]
    product=1
    for q in Q: product*=q
    assert product==64
    H=det_identity_coefficients(build_hashimoto(A))
    # det(I-uB)=(1-3u^2+3u^4-u^6)P(u)
```

```
    P=[]
    for k in range(43):
        P.append(H[k]+(3*P[k-2] if k>=2 else 0)-(3*P[k-4] if k>=4 else 0)+(P[k-6] if k>=6 else 0))
    assert P[37:]==[0]*6
    P=P[:37]; polynomials[name]=P
    assert sum(P)==0 and sum(k*c for k,c in enumerate(P))==7134
    for u in (Fraction(1,2),Fraction(2)):
        M=[[(int(i==j)*(1+u*u*Q[i])-u*A[i][j]) for j in range(n)] for i in range(n)]
        assert exact_det(M)==sum(c*u**k for k,c in enumerate(P))
    print(name, 'matching counts:',mc,'Ihara-Bass P coefficients:',P)

# Inversion swaps the two bipartite halves; it ANTICOMMUTES with gamma.
for i,u in enumerate(V_CHR):
    j=IDX_CHR[IOTA_MAP[u]]
    assert (1 if i<9 else -1)==-(1 if j<9 else -1)
    assert IOTA_MAP[IOTA_MAP[u]]==u
for i in range(18):
    for j in range(18):
        assert A_chr[i][j]*((1 if i<9 else -1)+(1 if j<9 else -1))==0

# Exact row/column-sum certificates for the ramp commutator at t=1.
# ||M||_2 <= sqrt(||M||_1 ||M||_infinity) <= 1.
for N in (6,54):
    ramp=[Fraction(i,2) for i in range(N)]
    L=[[(ramp[j]-ramp[i] if abs(i-j)==1 else Fraction(0)) for j in range(N)] for i in range(N)]
    assert max(sum(abs(x) for x in row) for row in L)<=1
    assert max(sum(abs(L[i][j]) for i in range(N)) for j in range(N))<=1
    assert ramp[-1]-ramp[0]==Fraction(N-1,2)
assert 4**6==4096

# Deliberately false historical claims are rejected, not silently certified.
assert traces_chr[12] != 192 and traces_tet[12] != 192
assert IDX_CHR[IOTA_MAP[V_A_chr[0]]] >= 9
print('Additional exact audit passed: group law, matchings, polynomial identities, grading and ramp
    bounds.')


with open("verification_values_a102.tex", "w", encoding="utf-8") as f:
    f.write("% Generated automatically by verify_a102.py (Blocks Z1--Z7)\n")
    for k_name, v_str in macros.items():
        f.write(f"\\newcommand{{\\{k_name}}}{{{v_str}}}\n")

print(f"VERIFICATION COMPLETE: All assertions in Blocks Z1--Z7 passed ({len(macros)} macros written).")
```

4 Execution output

```
chr matching counts: (1, 21, 180, 816, 2118, 3179, 2649, 1108, 189, 8) Ihara-Bass P coefficients: [1, 0, 3, 0, 6, 0,
    2, 0, -9, 0, -33, 0, -48, 0, -66, 0, -59, 0, -73, 0, -67, 0, -95, 0, -64, 0, -34, 0, 73, 0, 119, 0, 168, 0, 112,
     0, 64]
tet matching counts: (1, 21, 180, 814, 2096, 3092, 2497, 993, 155, 5) Ihara-Bass P coefficients: [1, 0, 3, 0, 6, 0,
    2, 0, -9, 0, -33, 0, -48, 0, -58, 0, -51, 0, -53, 0, -63, 0, -87, 0, -96, 0, -66, 0, 25, 0, 131, 0, 188, 0, 144,
     0, 64]
Additional exact audit passed: group law, matchings, polynomial identities, grading and ramp bounds.
VERIFICATION COMPLETE: All assertions in Blocks Z1--Z7 passed (43 macros written).
```