Quantum Behavior of Substituted-Chrysene Molecular Assemblies

Bipartite Transfer, Stark Response, and Registry-Gated Transport

Verification Code Companion

Charles D. Schaper, Ph.D.
Annals of the Chrysene Formalism, Volume 1, Number 4
Article 93, November 2026

1 Verified result and scope

The replacement suite executes **18,181 predicates in each of normal and optimized Python modes**: 17,093 exact-arithmetic predicates (including ten deliberately false negative controls) and 1,088 numerical predicates. Both modes produce identical results. The source uses explicit checks rather than Python assertions, so optimization cannot remove the verification.

The suite verifies the declared finite graph and chain models. It also supplies an *exact rational enclosure certificate* for the unperturbed polarizability, supported by the spectral lemmas explained below. It does not prove that a molecular device realizes the parameters, that broken inversion always produces a nonzero dipole, or that a finite-contact conductance ratio equals a bare Green-function ratio.

2 Coverage and corrected values

T1: Graph and matchings.
The declared 18-vertex graph has 21 undirected edges and a balanced bipartition. Independent recursive matching enumeration gives eight matchings. Fraction-based elimination gives $\det A = -64$. The exact inverse is checked on both sides. The determinant/matching relation is verified for this graph, not assumed for an arbitrary benzenoid.

T2: Paths and resolvent.
Powers certify three length-five y walks and one length-seven x walk, with no shorter walks. Deleting the respective terminals leaves four and one perfect matchings. The inverse entries have magnitudes 1/2 and 1/8, all same-sublattice blocks vanish, and the bare squared ratio is 16. Independent contacted inverses show ratios approaching 16 only at weak broadening: at normalized broadening 1 the ratio is 10.5625.

T3: Spectrum and response.
A Jacobi computation checks residuals, orthonormality, paired energies and uniform half-filled vertex charge. Its computed normalized values are

$$\begin{aligned} \Delta_{\mathrm{HL}} &= 1.040277576178548, \\ \alpha_{xx} &= 4.365841050491468, \qquad \alpha_{yy} = 6.343777755696013, \\ \alpha_{xy} &= -1.121153143544389, \quad \alpha_{xx}/\alpha_{yy} = 0.688208386015955. \end{aligned}$$

The HOMO–LUMO transition residues are approximately 0.55783698 and 0.26027055. Exact rational enclosures certify unequal diagonal response and unequal residues. An independent finite-difference ground-energy curvature agrees with the sum-over-states response within 2×10^{-4}. These values concern $H_0 = -A$, not all nonzero substituent potentials.

T4: Inversion and Stark examples.
Exact coordinate and graph inversion checks establish the support symmetries of D, T and asymmetries of I, Dm. The geometric vectors have squared lengths $37/4, 13/4$ and inner product $-7/4$. They are not themselves electronic dipoles. Separate gapped examples at site shifts 0.1, 0.5 and 1.0 have zero dipole on D, T and nonzero dipoles on I, Dm. Those examples do not prove a universal nonzero-dipole assertion.

T5: Chain scattering.
All contact words through 12 sites are checked at $\tau = 1/2$ for reversal, bounds and balance. Direct complex Green-matrix inversion checks the Fisher–Lee formula and $T + R = 1$ for all words through eight sites at $\tau = 1/2, 2/3$, with matched lead coupling. Perfect transmission is distinguished from scattering unitarity. Actual complement/reversal frame actions are checked on site words through length ten.

T6: Distribution.
Exact enumeration checks the shifted-binomial law for $N = 1, \ldots, 6$. Six sites have 32 contact fields and histogram $(10, 15, 6, 1)$ for $|\Delta m| = 0, 1, 2, 3$. The exact mean is $12814237/19536400 \simeq 0.6559$. These finite checks do not replace the manuscript's general asymptotic argument.

3 Exact rational spectral enclosure

Floating-point diagonalization is used only to suggest a basis. Its 18 approximate eigenvectors are rounded to 13 decimal places and regarded as *exact rational* columns of Q. The code constructs Q^{-1} in rational arithmetic, checks both inverse identities, and calculates $B = Q^{-1}HQ$ exactly. Each Gershgorin disk has rational center c_i and radius r_i. Their real intervals are disjoint and ordered. By the connected-component form of Gershgorin's theorem, each disk contains exactly one eigenvalue; similarity to the real symmetric H ensures all eigenvalues are real.

Let $\varepsilon = 10^{-7}$. For each approximate vector q_i, exact inequalities certify

$$(1 - \varepsilon/4)^2 < \|q_i\|^2 < (1 + \varepsilon/4)^2, \qquad \|Hq_i - c_i q_i\| < \varepsilon g_i/4,$$

where g_i is a lower bound on the distance from c_i to every other eigenvalue interval. The implemented upper residual bound is $18 \max_k |(Hq_i - c_i q_i)_k|$. Spectral decomposition then bounds the perpendicular component by $\varepsilon/4$. Choosing the sign of the isolated real eigenvector gives $\|q_i - v_i\| < \varepsilon$. This uses the norm bound and the triangle inequality; the eigenvector norm is not silently assumed to be exactly one.

For diagonal μ, the transition error is at most $3\varepsilon\|\mu\|$. Squaring the resulting intervals and dividing by the positive certified spectral-gap intervals gives rational lower and upper bounds for every term of the response sum. For y, rational coordinates $y/\sqrt{3}$ are used and the diagonal response is multiplied by 3. The resulting certified outer bounds are

$$4.365 < \alpha_{xx} < 4.367, \qquad 6.343 < \alpha_{yy} < 6.345.$$

The same construction gives disjoint HOMO–LUMO residue intervals. The basis and interval certificate are exported in `spectral_certificate.json`. The standard spectral lemmas in this explanation are not formalized in the Lean companion.

4 Reproduction and provenance

Run `python run_verification.py`. The runner validates the source SHA-256 pin, executes in isolated temporary directories in normal and optimized modes, compares every output byte, and writes `execution_audit.json`. The suite requires only Python's standard library. A direct run of `verify_a93.py` regenerates the JSON reports and numerical LaTeX macros.

The inherited script computed one polarizability tensor but exported different hardcoded constants. The replacement exports the actual computed values and checks exact enclosures. The earlier claim of independent Fisher–Lee checking was also unsupported by the inherited code; this implementation performs the stated matrix-inversion comparison. The corrected contact count is $2N-1$, not $2N-2$.

The standalone LaTeX source contains the identical Python source in Base64 form, emitted when compiled. Decode that file to recover the executable; the package supplies it directly as well. Typesetting does not execute the code. The complete readable listing follows.

5 Complete verification source

verify_a93.py

```
#!/usr/bin/env python3
"""Article 93 reproducible verification. Python standard library only.
Exact graph/chain predicates; exact rational spectral enclosures; numerical
finite-difference crosschecks separately labelled. Checks survive python -O.
No claim of chemical realization, universal dipole, or full analytic proof.
"""
from fractions import Fraction as F
from itertools import product
from math import sqrt, atan2, cos, sin, comb
from pathlib import Path
import json

CHECKS = {}
def check(ok, label, kind="exact"):
    CHECKS[kind] = CHECKS.get(kind, 0) + 1
    if not ok:
        raise RuntimeError("FAILED: " + label)
def mm(a,b):
    return [[sum(x*y for x,y in zip(row,col)) for col in zip(*b)] for row in a]
def eye(n):
    return [[int(i==j) for j in range(n)] for i in range(n)]
def inv(a):
    n=len(a); a=[[F(x) for x in row]+[F(i==j) for j in range(n)] for i,row in enumerate(a)]
    for c in range(n):
        p=next(i for i in range(c,n) if a[i][c]); a[c],a[p]=a[p],a[c]
        d=a[c][c]; a[c]=[x/d for x in a[c]]
        for i in range(n):
            if i!=c:
                d=a[i][c]; a[i]=[x-d*y for x,y in zip(a[i],a[c])]
    return [r[n:] for r in a]
def determinant(a):
    a=[[F(x) for x in r] for r in a]; out=F(1); n=len(a)
    for c in range(n):
        p=next((i for i in range(c,n) if a[i][c]),None)
        if p is None: return F(0)
        if p!=c: a[c],a[p]=a[p],a[c]; out=-out
        d=a[c][c]; out*=d
        for i in range(c+1,n):
            f=a[i][c]/d
            for j in range(c+1,n): a[i][j]-=f*a[c][j]
    return out
V_A = ['v1', 'v3', 'v5', 'v8', 'v10', 'v12', 'w2', 'w3', 'w6']
V_B = ['v2', 'v4', 'v6', 'v7', 'v9', 'v11', 'w1', 'w4', 'w5']
ADJ_LIST = {'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']}
COORDS_INT = {'v1': (-2, 2), 'v2': (-1, 3), 'v3': (1, 3), 'v4': (2, 2), 'v5': (4, 0), 'v6': (5, -1), 'v7
     ': (5, -3), 'v8': (4, -4), 'v9': (2, -4), 'v10': (1, -3), 'v11': (-1, -1), 'v12': (-2, 0), 'w1':
     (-1, 1), 'w2': (1, -1), 'w3': (1, 1), 'w4': (2, -2), 'w5': (2, 0), 'w6': (4, -2)}
VERTICES=V_A+V_B; IDX={v:i for i,v in enumerate(VERTICES)}; n=18
A=[[int(v in ADJ_LIST[u]) for v in VERTICES] for u in VERTICES]
H=[[-x for x in row] for row in A]
check(len(VERTICES)==len(set(VERTICES))==18,"18 vertices")
for i in range(n):
    for j in range(n):
        check(A[i][j]==A[j][i],"undirected adjacency")
        check(not A[i][j] or (i<9)!=(j<9),"bipartite edge")
check(sum(map(sum,A))==42,"21 edges")
def matchings(vertices):
    if not vertices: return 1
    u=vertices[0]
```

```
    return sum(matchings([w for w in vertices[1:] if w!=v])
               for v in vertices[1:] if A[u][v])
K=matchings(list(range(n))); detA=determinant(A)
check(K==8,"eight matchings"); check(detA==-64,"det A")
Ai=inv(A)
for p in [mm(A,Ai),mm(Ai,A)]:
    for i in range(n):
        for j in range(n): check(p[i][j]==int(i==j),"two-sided exact inverse")
x1,x2,y1,y2=[IDX[x] for x in ['v3','v9','v12','v6']]
check(Ai[y1][y2]==F(1,2),"y inverse sign/value")
check(Ai[x1][x2]==F(1,8),"x inverse sign/value")
check(matchings([i for i in range(n) if i not in [y1,y2]])==4,"y deletion matching")
check(matchings([i for i in range(n) if i not in [x1,x2]])==1,"x deletion matching")
check((Ai[y1][y2]/Ai[x1][x2])**2==16,"bare amplitude squared ratio")
for i in range(n):
    for j in range(n):
        if (i<9)==(j<9): check(Ai[i][j]==0,"same-sublattice inverse zero")
powers=[eye(n)]
for k in range(1,9): powers.append(mm(powers[-1],A))
for a,b,d,c in [(y1,y2,5,3),(x1,x2,7,1)]:
    for k in range(d): check(powers[k][a][b]==0,"shorter walk absent")
    check(powers[d][a][b]==c,"shortest walk count")
# The determinant equality is graph-specific; no universal benzenoid claim.
# Coordinates x and y/sqrt(3), in normalized coordinate units.
x=[F(COORDS_INT[v][0],2)-F(3,4) for v in VERTICES]
y=[F(COORDS_INT[v][1],2)+F(1,4) for v in VERTICES]
check(sum(x)==sum(y)==0,"centered coordinates")
perm=[next(j for j in range(n) if x[j]==-x[i] and y[j]==-y[i]) for i in range(n)]
for i in range(n):
    check(perm[perm[i]]==i,"inversion involution")
    for j in range(n): check(A[perm[i]][perm[j]]==A[i][j],"inversion graph symmetry")
Ds={'D':[6,12], 'T':[3,6,9,12], 'I':[3,6,12], 'Dm':[6]}
vectors={k:(sum(x[IDX['v'+str(l)]] for l in s),sum(y[IDX['v'+str(l)]] for l in s)) for k,s in Ds.items()}

check(vectors['D']==vectors['T']==(0,0),"D/T geometric sums")
check(vectors['I']==(F(-1,4),F(7,4)),"I geometric vector")
check(vectors['Dm']==(F(7,4),F(-1,4)),"Dm contrast vector")
metric=lambda u,v:u[0]*v[0]+3*u[1]*v[1]
check(metric(vectors['I'],vectors['I'])==F(37,4),"I squared length")
check(metric(vectors['Dm'],vectors['Dm'])==F(13,4),"Dm squared length")
check(metric(vectors['I'],vectors['Dm'])==F(-7,4),"cross product")
for name,s in Ds.items():
    support={IDX['v'+str(l)] for l in s}
    check(({perm[i] for i in support}==support)==(name in ['D','T']),"potential inversion criterion")

def jacobi(M):
    n=len(M); a=[[float(x) for x in r] for r in M]; q=eye(n)
    for it in range(200000):
        p,r=max(((i,j) for i in range(n) for j in range(i+1,n)),key=lambda ij:abs(a[ij[0]][ij[1]]))
        if abs(a[p][r])<1e-14: break
        theta=.5*atan2(2*a[p][r],a[r][r]-a[p][p]); c,s=cos(theta),sin(theta)
        pp,rr,pr=a[p][p],a[r][r],a[p][r]
        a[p][p]=c*c*pp-2*s*c*pr+s*s*rr
        a[r][r]=s*s*pp+2*s*c*pr+c*c*rr; a[p][r]=a[r][p]=0.
        for k in range(n):
            if k not in [p,r]:
                kp,kr=a[k][p],a[k][r]
                a[k][p]=a[p][k]=c*kp-s*kr; a[k][r]=a[r][k]=s*kp+c*kr
            qp,qr=q[k][p],q[k][r];q[k][p]=c*qp-s*qr;q[k][r]=s*qp+c*qr
    else: raise RuntimeError("Jacobi iteration limit")
    pairs=sorted((a[i][i],[q[k][i] for k in range(n)]) for i in range(n))
    return [p[0] for p in pairs],[p[1] for p in pairs]
E,Q=jacobi(H)
for i in range(n):
    for j in range(n):
        check(abs(sum(Q[i][k]*Q[j][k] for k in range(n))-int(i==j))<1e-11,"orthonormal eigenvectors","
     numerical")
    check(max(abs(sum(H[k][j]*Q[i][j] for j in range(n))-E[i]*Q[i][k]) for k in range(n))<1e-11,"eigen
```

```
    residual","numerical")
    check(abs(E[i]+E[-1-i])<1e-11,"paired eigenvalues","numerical")
    check(abs(2*sum(Q[o][i]**2 for o in range(9))-1)<1e-11,"half-filled vertex density","numerical")
def elem(a,op,b): return sum(u*v*w for u,v,w in zip(a,op,b))
def alpha(op1,op2):
    return sum(4*elem(Q[o],op1,Q[u])*elem(Q[o],op2,Q[u])/(E[u]-E[o]) for o in range(9) for u in range
    (9,18))
axx=float(alpha(x,x)); ayy=3*float(alpha(y,y)); axy=sqrt(3)*float(alpha(x,y))
gap=E[9]-E[8]
check(abs(gap-1.04027757617855)<1e-11,"gap","numerical")
check(axx<ayy and axx*ayy>axy**2,"positive anisotropic tensor","numerical")
resx=float(elem(Q[8],x,Q[9])**2); resy=3*float(elem(Q[8],y,Q[9])**2)
check(abs(resx-resy)>.01,"HOMO-LUMO unequal residues","numerical")

# Exact certificate: rationalized eigenbasis, exact similarity Gershgorin
# disks, residual isolation and interval propagation. Floating estimates
# only suggest a basis; all following certificate inequalities are rational.
q=[[F(f'{v:.13f}') for v in row] for row in Q]
U=list(map(list,zip(*q))); Ui=inv(U)
check(mm(U,Ui)==eye(n) and mm(Ui,U)==eye(n),"invertible rational eigenbasis")
B=mm(mm(Ui,H),U); centers=[B[i][i] for i in range(n)]
radii=[sum(abs(B[i][j]) for j in range(n) if j!=i) for i in range(n)]
lo=[centers[i]-radii[i] for i in range(n)]; hi=[centers[i]+radii[i] for i in range(n)]
for i in range(n-1): check(hi[i]<lo[i+1],"disjoint Gershgorin intervals")
check(hi[8]<0<lo[9],"nine occupied eigenvalues")
eps=F(1,10**7)
for i in range(n):
    normsq=sum(v*v for v in q[i])
    check((1-eps/4)**2<normsq<(1+eps/4)**2,"eigenvector norm bracket")
    separation=min(min(abs(centers[i]-lo[j]),abs(centers[i]-hi[j])) for j in range(n) if j!=i)
    residual=n*max(abs(sum(H[k][j]*q[i][j] for j in range(n))-centers[i]*q[i][k]) for k in range(n))
    check(residual<eps*separation/4,"eigenvector residual separation")
def square_interval(a,b): return (0 if a<=0<=b else min(a*a,b*b),max(a*a,b*b))
def certified_diagonal(op,factor=1):
    low=F(0); high=F(0); err=3*eps*max(map(abs,op))
    for o in range(9):
        for u in range(9,18):
            m=elem(q[o],op,q[u]); l,h=square_interval(m-err,m+err)
            gl=lo[u]-hi[o]; gh=hi[u]-lo[o]
            check(gl>0,"positive certified denominator")
            low+=4*factor*l/gh; high+=4*factor*h/gl
    return low,high
ix=certified_diagonal(x); iy=certified_diagonal(y,3)
check(F(4365,1000)<ix[0]<ix[1]<F(4367,1000),"alpha xx rigorous bracket")
check(F(6343,1000)<iy[0]<iy[1]<F(6345,1000),"alpha yy rigorous bracket")
check(ix[1]<iy[0],"rigorous static anisotropy")
# HOMO-LUMO residue intervals, again x and y/sqrt3 scaling.
rx=square_interval(elem(q[8],x,q[9])-3*eps*max(map(abs,x)),elem(q[8],x,q[9])+3*eps*max(map(abs,x)))
ry0=square_interval(elem(q[8],y,q[9])-3*eps*max(map(abs,y)),elem(q[8],y,q[9])+3*eps*max(map(abs,y)))
ry=(3*ry0[0],3*ry0[1])
check(rx[1]<ry[0] or ry[1]<rx[0],"rigorous unequal transition residues")

def energy(h): return 2*sum(jacobi(h)[0][:9])
base=energy(H); step=1e-3
for op,target in [(x,axx),(y,ayy/3)]:
    def shifted(f): return [[H[i][j]-(f*float(op[i]) if i==j else 0) for j in range(n)] for i in range(n)
    ]
    finite=-(energy(shifted(step))+energy(shifted(-step))-2*base)/step**2
    check(abs(finite-target)<2e-4,"energy-curvature independent check","numerical")
# Inversion guarantees no dipole in D/T when the half-filled gap is open.
# I/Dm nonzero results here are parameter examples, not a universal theorem.
examples={}
for delta in [.1,.5,1.0]:
    for name,s in Ds.items():
        support={IDX['v'+str(l)] for l in s}
        h=[[H[i][j]+(delta if i==j and i in support else 0) for j in range(n)] for i in range(n)]
        ee,qq=jacobi(h);check(ee[9]-ee[8]>.1,"example remains gapped","numerical")
        charge=[2*sum(qq[o][i]**2 for o in range(9)) for i in range(n)]
```

```
        mu=[sum(float(op[i])*charge[i] for i in range(n)) for op in [x,y]]
        if name in ['D','T']:check(max(map(abs,mu))<1e-10,"inversion dipole zero","numerical")
        else:check(max(map(abs,mu))>1e-4,"example permitted dipole","numerical")
        examples[f'{name}:{delta}']=mu

# Independent complex matrix inverse and Fisher-Lee chain response.
def cinv(m):
    n=len(m); a=[[complex(x) for x in r]+[complex(i==j) for j in range(n)] for i,r in enumerate(m)]
    for c in range(n):
        p=max(range(c,n),key=lambda i:abs(a[i][c])); a[c],a[p]=a[p],a[c]
        d=a[c][c]
        if abs(d)<1e-18: raise RuntimeError('singular complex solve')
        a[c]=[x/d for x in a[c]]
        for i in range(n):
            if i!=c:
                d=a[i][c]; a[i]=[x-d*y for x,y in zip(a[i],a[c])]
    return [r[n:] for r in a]
def dm(bits): return sum(bits[::2])-sum(bits[1::2])
def trans(bits,tau):
    lam=tau**abs(dm(bits));return 4/(lam+1/lam)**2
hist={d:0 for d in range(4)}
for N in range(1,7):
    for bits in product([0,1],repeat=2*N-1):
        d=dm(bits);check(dm(bits[::-1])==d,"reversal imbalance")
        check(trans(bits,F(1,2))==trans(bits[::-1],F(1,2)),"reversal transmission")
        check(0<trans(bits,F(1,2))<=1,"transmission bound")
        check((trans(bits,F(1,2))==1)==(d==0),"balance iff perfect transmission")
        if N==3:hist[abs(d)]+=1
        if N<=4:
            for tau in [F(1,2),F(2,3)]:
                sz=2*N;m=[[0j]*sz for _ in range(sz)]
                for k,b in enumerate(bits):m[k][k+1]=m[k+1][k]=float(tau**b)
                m[0][0]=m[-1][-1]=1j
                g=cinv(m);tv=4*abs(g[0][-1])**2;rv=abs(1-2j*g[0][0])**2
                check(abs(tv-float(trans(bits,tau)))<1e-11,"Fisher-Lee matrix inverse","numerical")
                check(abs(tv+rv-1)<1e-11,"unitarity with reflection","numerical")
    counts={k:comb(2*N-1,k+N-1) for k in range(-(N-1),N+1)}
    exact={k:0 for k in counts}
    for bits in product([0,1],repeat=2*N-1):exact[dm(bits)]+=1
    check(exact==counts,"shifted binomial law")
    check(sum(counts.values())==2**(2*N-1),"normalized uniform law")
check(list(hist.values())==[10,15,6,1],"six-site histogram")
mean=sum(trans(b,F(1,2)) for b in product([0,1],repeat=5))/32
check(mean==F(12814237,19536400),"six-site exact mean")
check(trans((1,0,1),F(1,2))==F(64,289),"SSH four")
check(trans((1,0,1,0,1),F(1,2))==F(256,4225),"SSH six")
# Physical-frame action represented on binary site labels: global complement,
# reversal, and their composition induce identity or reversal on contact bits.
def contacts(s): return tuple(a^b for a,b in zip(s,s[1:]))
for size in [2,4,6,8,10]:
    for sites in product([0,1],repeat=size):
        eta=contacts(sites)
        for image in [tuple(1-z for z in sites),sites[::-1],tuple(1-z for z in sites[::-1])]:
            check(trans(contacts(image),F(1,2))==trans(eta,F(1,2)),"actual frame-action invariance")
# Finite electrode broadening does not preserve the bare 16 ratio generally.
ratios=[]
for broad in [.001,.2,1.]:
    ts=[]
    for a,b in [(x1,x2),(y1,y2)]:
        m=[[complex(z) for z in row] for row in A]
        m[a][a]+=1j*broad;m[b][b]+=1j*broad
        ts.append(4*broad*broad*abs(cinv(m)[a][b])**2)
    ratios.append(ts[1]/ts[0])
check(abs(ratios[0]-16)<1e-4 and abs(ratios[-1]-16)>1,"bare versus contacted transport","numerical")

# Negative controls: each deliberately false proposition must be rejected.
negative=[('matching count 7',K==7),('determinant positive',detA==64),
 ('equal graph paths',powers[5][y1][y2]==powers[5][x1][x2]),
```

```
 ('same-sublattice nonzero',Ai[x1][y1]!=0),('inverse y wrong',Ai[y1][y2]==F(1,8)),
 ('old alpha xx',abs(axx-9.7346)<1e-3),('old alpha yy',abs(ayy-5.9015)<1e-3),
 ('old contact law',sum(hist.values())==2**4),('all fields perfect',mean==1),
 ('I inversion symmetry',{perm[IDX['v'+str(l)]] for l in Ds['I']}=={IDX['v'+str(l)] for l in Ds['I']})]
for name,wrong in negative:check(not wrong,'negative control: '+name)
values={'valVertices':18,'valEdges':21,'valSublatticeA':9,'valSublatticeB':9,
 'valKekuleCount':8,'valDetAchr':-64,'valDistY':5,'valPathsY':3,'valDistX':7,'valPathsX':1,
 'valKekuleY':4,'valKekuleX':1,'valGreenYFrac':'1/2','valGreenXFrac':'1/8','valConductanceRatio':16,
 'valCrossGreenA':0,'valCrossGreenB':0,'valHomoLumoGap':f'{gap:.4f}',
 'valAlphaXX':f'{axx:.4f}','valAlphaYY':f'{ayy:.4f}','valAlphaXY':f'{axy:.4f}','valAlphaRatio':f'{axx/
    ayy:.4f}',
 'valDipoleD':0,'valDipoleT':0,'valDipoleIX':'-1/4','valDipoleIY':'7/4','valDipoleNormSqI':'37/4',
 'valDipoleDmX':'7/4','valDipoleDmY':'-1/4','valDipoleNormSqDm':'13/4','valDipoleDotIDm':'-7/4',
 'valTransOrdered':1,'valTransSSHFour':'64/289','valTransSSHSix':'256/4225',
 'valCountDmZero':10,'valCountDmOne':15,'valCountDmTwo':6,'valCountDmThree':1,'valTransMeanEndoSix':f'{
    float(mean):.4f}'}
report={'article':93,'checks':CHECKS,'total_checks':sum(CHECKS.values()),'negative_controls':len(
    negative),
 'polarizability':{'xx':axx,'yy':ayy,'xy':axy,'ratio':axx/ayy,'certified_xx_outer':[4.365,4.367],'
    certified_yy_outer':[6.343,6.345]},
 'gap':gap,'residues':[resx,resy],'six_site_mean':str(mean),'histogram':hist,'parameter_dipoles':
    examples,
 'contacted_ratios':ratios,'scope':'Finite graph and chain calculations; exact rational spectral
    enclosure certificate; no chemical realization or full analytic Lean formalization.'}
Path('verification_values_a93.tex').write_text('% Generated from executed computations; normalized units
    .\n'+''.join('\\newcommand{\\'+k+'}{'+str(v)+'}\n' for k,v in values.items()))
Path('verification_report.json').write_text(json.dumps(report,indent=2,sort_keys=True)+'\n')
Path('spectral_certificate.json').write_text(json.dumps({'basis_columns':[[str(v) for v in r] for r in q
    ],
 'eigenvalue_intervals':[[str(l),str(h)] for l,h in zip(lo,hi)],'epsilon':str(eps),
 'alpha_xx_outer':list(map(str,(F(ix[0]*10**9//1,10**9),F(-((-ix[1]*10**9)//1),10**9)))),
 'alpha_yy_outer':list(map(str,(F(iy[0]*10**9//1,10**9),F(-((-iy[1]*10**9)//1),10**9)))),
 'residue_intervals':[list(map(str,rx)),list(map(str,ry))]},indent=2)+'\n')
print(json.dumps(report,sort_keys=True))
```