Verification Code Dossier

Article 99

Endogenous and Exogenous Sources of Encoded Chrysene Systems

Functional Selection, Adaptive Realization, and Engineered Emulation

Charles D. Schaper, Ph.D.

Volume 1, Number 4 November 2026

Certification Summary

The original scientific checker is retained with Article 99 identifiers. Fresh normal and optimized Python runs each pass 213 checks, including eight negative-control executions in seven categories. Numerical checks retain the stated tolerances; no new general theorem is inferred.

Publication integration

This companion belongs to the revised manuscript formerly numbered Article 102 and now designated Article 99. The renumbering and presentation update preserve the supplied scientific content. Execution evidence below was refreshed for the renamed source. The historical package remains the provenance source; this is not a new independent peer review.

1 Execution and reproducibility

The complete listing in Section 5 is identical to the accompanying executable file `verify_a99.py`. This LaTeX document is self-contained: it does not require that Python file to compile. Compile this document with pdfLaTeX in Overleaf. Python execution is separate from PDF compilation. The tested environment was:

- Python 3.12.14; NumPy 2.3.5; SciPy 1.17.0.
- Absolute numerical tolerance: 2×10^{-11}, except for explicitly strict counterexample margins.
- No random sampling, random seeds, network access, or downloaded data are used by the verification program.

With Python and the stated dependencies installed, execute:

```
python verify_a99.py --output results.json
```

The program stops with an assertion error on failure. A successful run writes a JSON object containing metadata and every named check. The companion `verification_output.txt` records the actual successful standard output. Other library versions can change numerical results and must be rerun; success is not asserted for untested environments.

SHA-256 of the executed Python source:
`9650324bec2aa8ed0bc3333c3dbf69bd`
`02e9d93c59b69c78d01a17c96c194bb4`

2 Coverage and observed results

Block	Checks	Principal coverage
Registry	14	All 64 orientation strings, 32 registry words, 20 reversal classes; robust law and rejection; finite minimax LP.
Information	20	KL disintegration with a zero target mass, Pythagorean and hierarchical identities, exact rejection law and geometric tails, microscopic excess.
Dynamics	66	Detailed balance, unchanged intra-fiber rates, spectral gap comparison, semigroup and output bounds, disconnected supports.
Trajectories	13	All 81 paths at four time indices of a three-state chain; event and soft-target Doob laws, zero rows, initial reweighting, quotient controls.
Approximation and kinetics	85	History-dependent path approximation, bounded rewards and costs, budget counterexample, binary threshold and engineered response.
Graphs	9	Actual 54-vertex box, 117 edges, 45 vertical edges; ranks over $\mathbb{F}_2$ and $\mathbb{F}_4$, cycle consistency and alphabet counts.
Channels	6	Kraus completeness, diagonal action, coherence removal, trace preservation, positivity and reliability contraction.
Total	213	All passed.

2.1 Exact arithmetic versus numerical checks

Python integers and `Fraction` implement exact orientation enumeration, quotient counts, rational selection and acceptance laws, geometric sums with their exact tails, rational path laws, history-dependent approximation witnesses and finite-field elimination. The finite-field implementation uses $\mathbb{F}_4 = \mathbb{F}_2[t]/(t^2 + t + 1)$, not integers modulo four. Incidence ranks are 45 on the vertical graph and 53 on the full graph over both fields; the full cycle-space dimension is $117 - 53 = 64$.

Logarithms, entropies, spectral gaps, matrix exponentials, linear programming and complex Kraus matrices are floating-point computations. They are explicitly tolerance-based, not interval-certified inequalities. The code compares reversible generators through symmetric similarity transforms and verifies symmetry before extracting eigenvalues. Positive and negative examples have margins substantially larger than the numerical tolerance. Exact rational equalities are tested by equality, not by floating-point tolerances.

2.2 Selected analytic benchmarks

The reversal quotient has eight singleton classes and twelve two-member classes. Under a uniform law on the 32 registry words these classes have probabilities 1/32 and 1/16, respectively, rather than 1/20. Uniform robust functional coverage has atom probabilities 1/20 on singleton classes and

1/40 on each member of a doubleton. Its information cost is

$$\frac{2}{5}\log\frac{8}{5}+\frac{3}{5}\log\frac{4}{5}=0.0541153209\ldots \text{ nats.}$$

The maximal rejection acceptance probability is 5/8 and the expected number of independent proposals is 8/5. Selecting a doubleton under the uniform reference costs four bits; selecting one specified member costs five bits.

2.3 Negative controls

The program deliberately checks that the following incorrect alternatives fail:

1. A uniform law on registry words induces a uniform law on reversal classes.
2. The Doob kernel alone gives the conditioned path law without reweighting the initial distribution.
3. The unchanged baseline generator has the desired nontrivially tilted stationary law.
4. Positive mass outside a reference support has finite relative entropy.
5. Approximate emulation preserves an identical budget without cost slack.
6. A modified edge registry with nonzero cycle sum is integrable.
7. Least KL under a nonuniform reference is automatically the maximum-entropy solution.

These are asserted counterexamples: their successful detection is part of a passing run.

2.4 Strong lumpability is sufficient, not necessary

The referee's counterexample is also checked exactly. Let $\pi(a)=\pi(b)=0$, $\pi(c)=1$, and take a terminal target $\{1\}$ at time two. Set

$$P_0=\begin{pmatrix}3/4 & 0 & 1/4\\ 1/4 & 0 & 3/4\\ 1/2 & 0 & 1/2\end{pmatrix},\qquad P_1=\begin{pmatrix}1/2 & 0 & 1/2\\ 1/2 & 0 & 1/2\\ 1/2 & 0 & 1/2\end{pmatrix}.$$

Although P_0 is not strongly lumpable, $h_1=h_0=(1/2,1/2,1/2)$. Thus the conditioning multiplier at time zero is identically one. A particular terminal objective can admit fiber-constant backward functions without universal strong lumpability. This check guards the manuscript's corrected distinction between a sufficient hypothesis and a necessary condition.

3 Relation to the manuscript and limitations

The registry and information blocks correspond to the selection results on least-information lifts, rejection realization, reversal classes and robust coverage. The dynamics and trajectory blocks correspond to rate realization, quantitative convergence, zero-target restrictions, event conditioning, terminal-law conditioning and quotient control. The approximation block corresponds to finite-horizon error, budget transfer and binary kinetic advantage. The graph and channel blocks correspond to coefficient-field registry integration and the stochastic-channel lift.

The tests use specified finite examples rather than quantification over all admissible spaces, controls, environments, kernels or molecular systems. The minimax solver checks a finite 20-output problem, not a general minimax theorem. The semigroup tests inspect stated times and initial states, not every time or initial law. The graph tests check the specified box and illustrative cycles,

while the manuscript proves the general integration criterion. The program does not establish chemical feasibility, thermodynamic optimality, realizable sensing, a physical cost model, or an origin-independent superiority claim. A passing run is evidence of computational consistency within this stated scope.

4 Executed output

```
PASS registry: 14 checks
PASS information: 20 checks
PASS dynamics: 66 checks
PASS trajectories: 13 checks
PASS approximation_and_kinetics: 85 checks
PASS graphs: 9 checks
PASS channels: 6 checks
ALL PASS: 213 checks; results written to verification_report.json
{
  "python": "3.12.14",
  "numpy": "2.3.5",
  "scipy": "1.17.0",
  "absolute_tolerance": 2e-11,
  "script_sha256": "9650324bec2aa8ed0bc3333c3dbf69bd02e9d93c59b69c78d01a17c96c194bb4",
  "scope": "Finite exact arithmetic/enumeration and floating-point regression checks; not a
    universal proof or physical validation.",
  "exactness": "Checks named exact, registry counts, rational acceptance, and finite-field ranks
    use Python integer/Fraction arithmetic. Logarithms, spectral gaps, exponentials, LP, and Kraus
    matrices use floating point."
}
```

5 Complete executable listing

verify_a99.py

```
#!/usr/bin/env python3
"""Article 99: reproducible finite-example checks, not universal proofs.
Run: python verify_a99.py --output results.json
Dependencies: numpy, scipy. All tolerances are explicit; no randomized tests.
"""
import argparse
import hashlib
import itertools as it
import json
import math
import platform
from fractions import Fraction as F
from pathlib import Path
import numpy as np
import scipy
from scipy.linalg import expm, eigvalsh
from scipy.optimize import linprog

TOL = 2e-11
RESULTS = []

def check(name, condition, detail=''):
    if not bool(condition):
        raise AssertionError(f'{name}: {detail}')
    RESULTS.append({'check': name, 'passed': True, 'detail': str(detail)})

def close(name, x, y, tol=TOL):
    error = float(np.max(np.abs(np.asarray(x)-np.asarray(y))))
    check(name, error <= tol, f'max absolute error={error:.4g}; tolerance={tol:g}')

def tv(p, q):
    return float(np.abs(np.asarray(p)-np.asarray(q)).sum()/2)

def kl(p, q):
    return sum(float(a)*math.log(float(a/b)) if b else math.inf
               for a,b in zip(p,q) if a)

def push(p, labels, m):
    return [sum(p[i] for i,y in enumerate(labels) if y==j) for j in range(m)]

def lift(mu, labels, r):
    q=push(mu,labels,len(r))
    return [r[y]*mu[i]/q[y] for i,y in enumerate(labels)]

def registry():
    words=list(it.product(range(2), repeat=5))
    orientation=list(it.product(range(2), repeat=6))
    agreement=lambda s: tuple(s[i]^s[i+1] for i in range(5))
    check('registry_XOR_encoding',agreement((0,0,1,1,0,1))==(0,1,0,1,1))
    counts={w:sum(agreement(s)==w for s in orientation) for w in words}
    check('registry_orientation_fibers',set(counts.values())=={2})
    orbits=sorted({min(w,w[::-1]) for w in words})
    labels=[orbits.index(min(w,w[::-1])) for w in words]
    sizes=[labels.count(j) for j in range(len(orbits))]
```

```
    check('registry_32_words_20_classes',len(words)==32 and len(orbits)==20)
    check('registry_8_singletons_12_pairs',sizes.count(1)==8 and sizes.count(2)==12)
    mu=[F(1,32)]*32; q=push(mu,labels,20)
    check('negative_control_uniform_quotient',q!=[F(1,20)]*20)
    r=[F(1,20)]*20; p=lift(mu,labels,r)
    check('robust_atom_weights',all(p[i]==F(1,20*sizes[y]) for i,y in enumerate(labels)))
    cost=F(2,5)*math.log(1.6)+F(3,5)*math.log(.8)
    close('robust_KL',kl(p,mu),cost)
    close('pair_vs_microstate_bits',[-math.log2(float(F(2,32))),-math.log2(float(F(1,32)))
    ],[4,5])
    c=min(q[j]/r[j] for j in range(20)); a=[c*r[j]/q[j] for j in range(20)]
    check('robust_rejection_exact',c==F(5,8) and 1/c==F(8,5))
    check('robust_rejection_acceptance',all(a[j]==(1 if sizes[j]==1 else F(1,2)) for j in
    range(20)))
    check('robust_rejection_law', [mu[i]*a[y]/c for i,y in enumerate(labels)]==p)
    # Linear-program variables: 20 output probabilities followed by worst coverage.
    obj=np.r_[np.zeros(20),-1.]
    Aub=np.c_[-np.eye(20),np.ones(20)]
    sol=linprog(obj,A_ub=Aub,b_ub=np.zeros(20),A_eq=[np.r_[np.ones(20),0]],b_eq=[1],bounds
    =[(0,None)]*21,method='highs')
    check('minimax_LP_success',sol.success)
    close('minimax_LP_value_and_uniformity',sol.x,np.r_[np.full(20,.05),.05])
    check('minimax_deterministic_zero',min([1]+[0]*19)==0)


def information():
    mu=[F(i,21) for i in range(1,7)]; lab=[0,0,1,1,2,2]
    r=[F(2,5),F(3,5),F(0)]; q=push(mu,lab,3); star=lift(mu,lab,r)
    p=[F(3,10),F(1,10),F(1,5),F(2,5),F(0),F(0)]
    check('KL_target_zero_mass_exact',push(p,lab,3)==r and push(star,lab,3)==r)
    cond=sum(float(r[y])*kl([p[i]/r[y] for i in range(6) if lab[i]==y],
                            [mu[i]/q[y] for i in range(6) if lab[i]==y])
             for y in range(3) if r[y])
    close('KL_chain_zero_target',kl(p,mu),kl(r,q)+cond)
    close('KL_Pythagorean',kl(p,mu),kl(p,star)+kl(star,mu))
    close('KL_lift_minimum_value',kl(star,mu),kl(r,q))
    for y in range(3):
        e=[F(int(j==y)) for j in range(3)]; py=lift(mu,lab,e)
        close(f'class_cost_{y}',kl(py,mu),-math.log(float(q[y])))
        for i in range(6):
            if lab[i]==y:
                delta=[F(int(k==i)) for k in range(6)]
                close(f'microstate_excess_{i}',kl(delta,mu)-kl(py,mu),-math.log(float(mu[i]/q[
    y])))
    psi=[0,0,1]; s=push(r,psi,2); qb=push(q,psi,2)
    excess=sum(float(s[z])*kl([r[y]/s[z] for y in range(3) if psi[y]==z],
                              [q[y]/qb[z] for y in range(3) if psi[y]==z]) for z in range(2) if
     s[z])
    close('nested_quotient_chain',kl(r,q),kl(s,qb)+excess)
    c=min(q[y]/r[y] for y in range(3) if r[y]); a=[c*r[y]/q[y] for y in range(3)]
    check('rejection_rational_exact',sum(mu[i]*a[y] for i,y in enumerate(lab))==c and [mu[i]*a
    [y]/c for i,y in enumerate(lab)]==star)
    check('rejection_optimality_saturation',max(a)==1 and max([F(101,100)*x for x in a])>1)
    # Exact finite sums including tails, not truncation approximations.
    n=25; probs=[(1-c)**(k-1)*c for k in range(1,n+1)]
    check('geometric_mass_including_tail',sum(probs)+(1-c)**n==1)
    check('geometric_mean_including_tail',sum(F(k)*probs[k-1] for k in range(1,n+1))+(1-c)**n
    *(n+1/c)==1/c)
    check('negative_control_forbidden_support',math.isinf(kl([F(1,2),F(1,2)],[F(1),F(0)])))
```

```
    # Nonuniform conditional baseline is not the entropy maximizer.
    H=lambda z:-sum(float(a)*math.log(float(a)) for a in z if a)
    check('negative_control_nonuniform_entropy',H([F(1,2),F(1,2)])>H([F(1,3),F(2,3)]))

def generator(off):
    Q=np.array(off,float).copy(); np.fill_diagonal(Q,0)
    np.fill_diagonal(Q,-Q.sum(axis=1)); return Q

def spectral_gap(Q,pi):
    S=np.sqrt(pi)[:,None]*Q/np.sqrt(pi)[None,:]
    close('reversible_similarity_symmetric',S,S.T)
    return float(-eigvalsh(S)[-2])

def dynamics():
    mu=np.array([.1,.2,.3,.4]); lab=[0,0,1,1]; r=[.7,.3]
    C=np.array([[0,.3,0,.2],[.3,0,.4,0],[0,.4,0,.5],[.2,0,.5,0]])
    Q=generator(C/mu[:,None]); star=np.array(lift(mu,lab,r)); w=star/mu
    Qt=generator(np.maximum(Q,0)*np.minimum(1,w[None,:]/w[:,None]))
    close('baseline_detailed_balance',mu[:,None]*Q,(mu[:,None]*Q).T)
    close('tilted_rows',Qt.sum(1),0)
    close('tilted_detailed_balance',star[:,None]*Qt,(star[:,None]*Qt).T)
    close('tilted_stationarity',star@Qt,0)
    for x,z in it.product(range(4),repeat=2):
        if x!=z and lab[x]==lab[z]: close(f'within_fiber_rate_{x}_{z}',Qt[x,z],Q[x,z])
    check('negative_control_unchanged_rates',np.linalg.norm(star@Q)>0.1)
    gap=spectral_gap(Q,mu); gapt=spectral_gap(Qt,star)
    check('spectral_gap_comparison',gapt+TOL>=min(w)/max(w)*gap,f'actual={gapt}; lower={min(w)
    /max(w)*gap}')
    for i in range(4):
        p0=np.eye(4)[i]; chi=sum((p0-star)**2/star)
        for t in [0,.02,.1,.3,1,3]:
            pt=p0@expm(t*Qt); bd=.5*math.exp(-gapt*t)*math.sqrt(chi)
            check(f'semigroup_TV_{i}_{t}',tv(pt,star)<=bd+TOL)
            check(f'output_contraction_{i}_{t}',tv(push(pt,lab,2),r)<=tv(pt,star)+TOL)
    # Remove vertex 2 from a 5-vertex path. Support is two components.
    C5=np.diag(np.ones(4),1)+np.diag(np.ones(4),-1)
    S=[0,1,3,4]; desired=np.array([.1,.2,.3,.4]); Qs=generator(C5[np.ix_(S,S)]/desired[:,None
    ])
    pbad=np.array([.5,.5,0,0]); pgood=np.array([.3,0,.7,0])
    for t in [0,.2,2,20]:
        close(f'disconnected_component_mass_{t}',(pbad@expm(t*Qs))[:2].sum(),1)
    check('disconnected_wrong_mass_no_convergence',tv(pbad@expm(20*Qs),desired)>.69)
    close('disconnected_matching_mass_converges',pgood@expm(20*Qs),desired)


def pathlaw(nu, kernels):
    n=len(nu); paths=list(it.product(range(n),repeat=len(kernels)+1)); out=[]
    for path in paths:
        p=nu[path[0]]
        for t,K in enumerate(kernels):p*=K[path[t]][path[t+1]]
        out.append(p)
    return paths,out

def doob(nu,kernels,g):
    h=[None]*(len(kernels)+1);h[-1]=g
    for t in range(len(kernels)-1,-1,-1):
        h[t]=[sum(kernels[t][x][z]*h[t+1][z] for z in range(len(nu))) for x in range(len(nu))]
    norm=sum(nu[x]*h[0][x] for x in range(len(nu)))
    newnu=[nu[x]*h[0][x]/norm for x in range(len(nu))]
```

```
    newK=[]
    for t,K in enumerate(kernels):
        newK.append([[K[x][z]*h[t+1][z]/h[t][x] if h[t][x] else F(int(x==z))
                      for z in range(len(nu))] for x in range(len(nu))])
    return newnu,newK,h,norm

def trajectories():
    nu=[F(1,2),F(1,3),F(1,6)]
    K=[[F(1,2),F(1,2),F(0)],[F(0),F(1,2),F(1,2)],[F(0),F(0),F(1)]]
    kernels=[K,K,K]; paths,base=pathlaw(nu,kernels)
    newnu,newK,h,a=doob(nu,kernels,[F(1),F(1),F(0)])
    _,cond=pathlaw(newnu,newK); expected=[p/a if path[-1] in [0,1] else F(0) for path,p in zip
    (paths,base)]
    check('Doob_event_full_path_exact',cond==expected and sum(cond)==1)
    check('Doob_zero_rows_present',any(v==0 for row in h for v in row))
    check('Doob_unreachable_zero_rows',all(not p or all(h[t][x]>0 for t,x in enumerate(path))
    for path,p in zip(paths,cond)))
    _,wrong=pathlaw(nu,newK)
    check('negative_control_missing_initial_reweight',tv(np.array(wrong,float),np.array(cond,
    float))>.1)
    close('Doob_event_KL',kl(cond,base),-math.log(float(a)))
    successes=[i for i,p in enumerate(cond) if p]
    R=[F(0)]*len(paths);R[successes[0]]=F(1,3);R[successes[-1]]=F(2,3)
    close('Doob_Pythagorean',kl(R,base),kl(R,cond)-math.log(float(a)))
    i=successes[0];close('Doob_history_excess',-math.log(float(base[i]))+math.log(float(a)),-
    math.log(float(cond[i])))
    terminal=push(base,[p[-1] for p in paths],3); target=[F(1,4),F(3,4),F(0)]
    g=[target[j]/terminal[j] for j in range(3)]
    nn,kk,hh,norm=doob(nu,kernels,g);_,soft=pathlaw(nn,kk)
    check('Doob_soft_path_exact',norm==1 and soft==[p*g[path[-1]] for path,p in zip(paths,base
    )])
    check('Doob_soft_terminal_exact',push(soft,[p[-1] for p in paths],3)==target)
    close('Doob_soft_KL',kl(soft,base),kl(target,terminal))
    # Lumpable blocks {0,1},{2,3}.
    L=[[F(1,4),F(1,4),F(1,3),F(1,6)],
       [F(1,8),F(3,8),F(1,8),F(3,8)],
       [F(1,6),F(1,6),F(1,3),F(1,3)],
       [F(1,4),F(1,12),F(1,2),F(1,6)]]
    _,_,lh,_=doob([F(1,4)]*4,[L,L],[F(0),F(0),F(1),F(1)])
    check('lumpable_h_fiber_constant',all(row[0]==row[1] and row[2]==row[3] for row in lh))
    # Not lumpable: rows 0 and 1 have unequal probability of block {2}.
    # Yet probability of the chosen target block {3} is identical, so its h is constant.
    N=[[F(1,4),F(1,4),F(1,4),F(1,4)],
       [F(1,2),F(1,8),F(1,8),F(1,4)],
       [F(0),F(0),F(1),F(0)], [F(0),F(0),F(0),F(1)]]
    _,_,nh,_=doob([F(1,4)]*4,[N],[F(0),F(0),F(0),F(1)])
    check('nonlumpability_not_necessary_for_target',N[0][2]!=N[1][2] and nh[0][0]==nh[0][1])
    P0=[[F(3,4),F(0),F(1,4)],[F(1,4),F(0),F(3,4)],[F(1,2),F(0),F(1,2)]]
    P1=[[F(1,2),F(0),F(1,2)]]*3
    _,wk,wh,_=doob([F(1,3)]*3,[P0,P1],[F(0),F(0),F(1)])
    check('referee_nonlumpable_target_witness_exact',P0[0][2]!=P0[1][2] and wh[:2]==[[F(1,2)
    ]*3]*2 and wk[0]==P0)


def history_law(nu, transition, T):
    d={(x,):p for x,p in enumerate(nu)}
    for t in range(T):
        d={h+(z,):p*transition(h)[z] for h,p in d.items() for z in range(2)}
    return d
```

```
def approximation_and_kinetics():
    nu=[F(2,5),F(3,5)]; alt=[F(1,2),F(1,2)]
    def kernel(h):
        a=F(2+sum(h),8);return [a,1-a]
    def emulator(h):
        a=kernel(h)[0]+F((-1)**sum(h),20);return [a,1-a]
    T=3;P=history_law(nu,kernel,T);R=history_law(alt,emulator,T)
    deltas=[F(1,10)]
    for t in range(1,T+1):
        deltas.append(max(abs(kernel(h)[0]-emulator(h)[0]) for h in it.product(range(2),repeat
    =t)))
    delta=min(F(1),sum(deltas)); pathTV=sum(abs(P[h]-R[h]) for h in P)/2
    check('history_TV_telescoping_exact',pathTV<=delta)
    # Observation is last bit, reward is bounded by M=3, cost by Mc=5.
    op=push(list(P.values()),[h[-1] for h in P],2);orr=push(list(R.values()),[h[-1] for h in R
    ],2)
    check('history_observation_contraction_exact',sum(abs(x-y) for x,y in zip(op,orr))/2<=
    pathTV)
    rewards={h:F(3*sum(h),len(h)) for h in P};cost={h:F(5*int(h[0]!=h[-1])) for h in P}
    EP=lambda v:sum(P[h]*v[h] for h in P);ER=lambda v:sum(R[h]*v[h] for h in R)
    check('reward_transfer_exact',ER(rewards)>=EP(rewards)-3*delta)
    check('cost_slack_transfer_exact',ER(cost)<=EP(cost)+5*delta)
    # Cost C=1_{state=1}: exact change TV=epsilon increases a zero budget.
    eps=F(1,10); base=[F(1),F(0)]; engineered=[1-eps,eps]; C=[F(0),F(1)]
    original_cost=sum(a*c for a,c in zip(base,C)); new_cost=sum(a*c for a,c in zip(engineered,
    C))
    budget=F(0); discrepancy=sum(abs(a-b) for a,b in zip(base,engineered))/2
    check('negative_control_same_budget',original_cost<=budget<new_cost and new_cost<=budget+
    discrepancy)
    # Environment-by-environment witnesses imply the finite minimax inequality.
    vals=[F(4,5),F(3,5)];approx=[F(3,4),F(11,20)]
    check('finite_minimax_transfer_exact',min(approx)>=min(vals)-F(1,20))
    for alpha,beta in [(3.,1.),(2.,.2),(1.1,1.)]:
        threshold=math.log(2*alpha/(alpha-beta))/(alpha+beta)
        G=generator([[0,alpha],[beta,0]])
        for p0 in [0.,.2,.6,1.]:
            for t in [0.,threshold,threshold+.1,threshold+2]:
                formula=alpha/(alpha+beta)+(p0-alpha/(alpha+beta))*math.exp(-(alpha+beta)*t)
                actual=(np.array([1-p0,p0])@expm(t*G))[1]
                close(f'binary_kinetics_{alpha}_{p0}_{t:.4f}',actual,formula)
                if t>threshold:check(f'binary_strict_advantage_{alpha}_{p0}_{t:.4f}',actual
    >.5)
        close(f'binary_threshold_equality_{alpha}',(np.array([1.,0.])@expm(threshold*G))
    [1],.5)
        # Independently form both demand-specific engineered generators by relabeling.
        swap=np.array([[0.,1.],[1.,0.]])
        engineered_demand_zero=generator([[0,beta],[alpha,0]])
        close(f'engineered_generator_recovery_{alpha}',expm(.7*engineered_demand_zero),
    swap@expm(.7*G)@swap)
    check('binary_uninformed_grid_bound',all(min(a,1-a)<=F(1,2) for a in [F(k,100) for k in
    range(101)]))


def fmul(a,b):
    # F4 = F2[t]/(t^2+t+1); integer bits encode polynomial coefficients.
    result=0
    while b:
        if b&1:result^=a
```

```
        b>>=1;a<<=1
        if a&4:a^=7
    return result

def rank_field(A,q):
    A=[list(map(int,row)) for row in A];m=len(A);n=len(A[0]);rank=0
    mul=(lambda a,b:a*b%2) if q==2 else fmul
    for col in range(n):
        pivot=next((i for i in range(rank,m) if A[i][col]),None)
        if pivot is None:continue
        A[rank],A[pivot]=A[pivot],A[rank]
        inv=next(x for x in range(1,q) if mul(A[rank][col],x)==1)
        A[rank]=[mul(x,inv) for x in A[rank]]
        for i in range(m):
            if i!=rank and A[i][col]:
                a=A[i][col];A[i]=[x^mul(a,y) for x,y in zip(A[i],A[rank])]
        rank+=1
        if rank==m:break
    return rank

def graphs():
    V=list(it.product(range(3),range(3),range(6)));index={v:i for i,v in enumerate(V)}
    E=[];vertical=[]
    for v in V:
        for axis,limit in enumerate([3,3,6]):
            if v[axis]+1<limit:
                z=list(v);z[axis]+=1;e=(index[v],index[tuple(z)]);E.append(e)
                if axis==2:vertical.append(e)
    check('actual_box_counts',len(V)==54 and len(E)==117 and len(vertical)==45)
    def incidence(edges):
        return [[int(i==x or i==z) for i in range(len(V))] for x,z in edges]
    A=incidence(E);Av=incidence(vertical)
    for q in [2,4]:
        rf=rank_field(A,q);rv=rank_field(Av,q)
        check(f'graph_rank_F{q}',rf==53 and rv==45 and len(E)-rf==64)
        # Explicit four-edge square, coboundary cycle sum vanishes in characteristic 2.
        square=[index[v] for v in [(0,0,0),(1,0,0),(1,1,0),(0,1,0)]]
        s=[(3*i+1)%q for i in range(54)]
        eta=[s[square[i]]^s[square[(i+1)%4]] for i in range(4)]
        total=0
        for e in eta:total^=e
        check(f'cycle_integrability_F{q}',total==0)
        eta[0]^=1;bad=0
        for e in eta:bad^=e
        check(f'negative_control_nonintegrable_F{q}',bad!=0)
    check('F4_multiplication_field',all(any(fmul(a,b)==1 for b in range(1,4)) for a in range
    (1,4)))
    check('registry_alphabet_counts',2**5==32 and 4**5==1024)


def channels():
    K=np.array([[.5,.25,.25],[.2,.5,.3],[.1,.2,.7]])
    # K[x,z] means K(z|x); both domain and codomain are X here.
    L=[]
    for x,z in it.product(range(3),repeat=2):
        A=np.zeros((3,3),complex);A[z,x]=math.sqrt(K[x,z]);L.append(A)
    close('Kraus_completeness',sum(A.conj().T@A for A in L),np.eye(3))
    channel=lambda rho:sum(A@rho@A.conj().T for A in L)
    p=np.array([.2,.3,.5]);close('Kraus_diagonal_lift',channel(np.diag(p)).real,np.diag(p@K))
```

```
    v=np.array([1,1j,2],complex)/math.sqrt(6);rho=np.outer(v,v.conj());out=channel(rho)
    close('Kraus_coherence_erasure',out,np.diag(np.real(np.diag(rho))@K))
    close('Kraus_trace_preservation',np.trace(out).real,1)
    check('Kraus_positive_output',eigvalsh(out).min()>=-TOL)
    p2=np.array([.3,.4,.3]);reward=np.array([.1,.9,.6])
    check('common_channel_reliability_bound',abs((p@K-p2@K)@reward)<=tv(p,p2)+TOL)


def main():
    parser=argparse.ArgumentParser();parser.add_argument('--output',default='results.json');
    args=parser.parse_args()
    for block in [registry,information,dynamics,trajectories,approximation_and_kinetics,graphs
    ,channels]:
        start=len(RESULTS);block();print(f'PASS {block.__name__}: {len(RESULTS)-start} checks'
    )
    metadata={'python':platform.python_version(),'numpy':np.__version__,'scipy':scipy.
    __version__,
              'absolute_tolerance':TOL,'script_sha256':hashlib.sha256(Path(__file__).
    read_bytes()).hexdigest(),
              'scope':'Finite exact arithmetic/enumeration and floating-point regression
    checks; not a universal proof or physical validation.',
              'exactness':'Checks named exact, registry counts, rational acceptance, and
    finite-field ranks use Python integer/Fraction arithmetic. Logarithms, spectral gaps,
    exponentials, LP, and Kraus matrices use floating point.'}
    report={'metadata':metadata,'passed':len(RESULTS),'failed':0,'checks':RESULTS}
    Path(args.output).write_text(json.dumps(report,indent=2)+'\n')
    print(f'ALL PASS: {len(RESULTS)} checks; results written to {args.output}')
    print(json.dumps(metadata,indent=2))

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