ACF 90 — Verification Dossier

The Source Duality

Endogenous and Exogenous Preparation of the Substituted-Chrysene Stock,
Mixed Arm Words, and the Persistence Semigroup

Charles D. Schaper

Annals of the Chrysene Formalism, Vol. 1, Issue 4, November 2026

Audited companion prepared 4 November 2026

Executed verification and scope

Item	Recorded result
Mathematical checks per run	71
Macro-name uniqueness checks per run	82
Total checks per run	153
Execution modes	Normal and Python `-O`
Scientific outputs	Byte-identical
Deliberate-defect runs rejected	10 of 10
Generated publication macros	82
Runtime	Python 3.12.14; standard library only

1 Purpose and limits

This companion supplies reproducible finite calculations for Article 90, *The Source Duality*. The complete executable and audit driver are printed below and embedded in both this single TeX source and its digital PDF. The verified objects are rational identities, finite arm-pattern enumerations, finite lattice counts and selected numerical examples. A successful run does not prove all statements of the manuscript.

The controlling executable is `verify_a90.py`. It is based on the revised Python source supplied under the misleading filename `verify_a90.tex`, with explicit corrections and additional checks. The original files have been preserved. `SOURCE_PIN.json` records their hashes and the hashes of the executable files in this release. Hashes establish byte integrity, not scientific correctness or authorship.

2 Orthogonal-review revision concordance

The revised manuscript fixes a coherent arm-representative convention, proves covariance under registry-coordinate changes, separates placed preparation from zero-flip retention, and states the positive-rate hypotheses for eventual degree-4 absorption. It also corrects the chemical naming of the hydroxyethoxy arm, treats conversion and contact-energy laws as explicit models, and qualifies the thermodynamic work interpretation. None of these revisions changes the finite numerical targets checked by this companion. The executable and its finite certificate count are unchanged. The new general covariance proof and the expanded finite-chain argument are manuscript proofs, not additional machine-checked theorems. Chemical synthesis and physical realization remain outside the executable's scope.

3 Coverage of the nine verification blocks

Block	Executable coverage and qualification
Y1	Exact cascade probabilities, normalization, stock ratio, rate k_3, demand inequality, efficiencies and rounded flux. This does not mechanize the general Jacobian orthogonality theorem.
Y2	Explicit classes at $m = 2, 3$; fixed and visible patterns; $D(eh)$, $T(eeee)$ and initiator-cap coincidence counts. Nonuniform quotient masses are normalized using the support stabilizer.
Y3	Binary entropy and Boltzmann examples at the stated contact energies. These are floating-point checks, not interval-certified transcendental inequalities.
Y4	Exact rational arm efficiencies at five inputs and their rounded displays. No universal Pareto optimality theorem is proved by these samples.
Y5	Cooperativity and initiator-contact frequencies at three inputs; selected information-gain values.
Y6	Hexamer species costs at two ratios and finite registry-count arithmetic.
Y7	Buffered mean-return arithmetic, distinct stack- and column-start times, two exact first-step cap-mean solves, and corrected finite-time retention samples. The stochastic-process theorems remain mathematical arguments in the manuscript.
Y8	Numbers 4, 4096, 2.50 and 26.50 used in the quotient-dimension and conditional spectral-witness examples. No Dirac commutator, spectral-distance finiteness or metric equality is tested.
Y9	The $3 \times 3 \times 6$ box has 54 nodes, 72 planar bridges, 45 stacking contacts, 72 free arms and 4 complete sites. All nine planar free-terminal patterns and their reflections are checked. Physical connector locking is a separate modelling assumption.

4 Corrections established by this audit

The legacy `verify_duality.py` fails at its first entropy-gain assertion. The revised supplied generator runs, but several manuscript fallbacks and manually emitted constants are stale. This release uses recomputed values:

Quantity	Earlier display	Corrected display
$1 - h_2((1+e^2)^{-1})$	0.4728	0.4729
$1 - h_2((1+e^4)^{-1})$	0.8701	0.8700
$\eta_{\mathrm{arm}}(0.75)$	0.9228	0.9286
$\eta_{\mathrm{arm}}(0.50)$	0.6961	0.7117
I–I gain at unit bias	0.1887	0.1601
$f_I(0.1)^2$	2.5×10^{-7}	2.39×10^{-7}
$f_I(1)^2$	2.38×10^{-5}	2.37×10^{-5}
$f_I(10)^2$	2.16×10^{-3}	2.17×10^{-3}

Here $\rho^* = 42354/208 = 21177/104 = 1629/8 = 203.625$. The last three values use this exact ratio, rather than an unstated rounded stock ratio. The complete emitted macro set is the publication input; the former claim of 57 macros is replaced by the actual count 82.

4.1 Return times and finite-time retention

At $(r, a, d) = (0.001, 5, 0.05)$, the stack-start mean is $(1 + a/d)/r = 101000$, whereas the column-start mean is $1/d + (1 + a/d)/r = 101020$. These are emitted separately. For the cap conversion chain

with transient states C_3, S_3, B_3, B_4, S_4 and absorbing state C_4, the checker solves $(-Q)\tau = \mathbf{1}$ in exact rational arithmetic and verifies, at two parameter sets,

$$\tau_{C_3} = \frac{1}{d_3} + \frac{1 + a_3/d_3}{r_3}\left(1 + \frac{e_3}{k_4}\right) + \frac{1}{k_4} + \frac{1}{e_4} + \frac{1 + r_4/e_4}{a_4}.$$

This is a finite evaluation of the first-step formula, not a formal proof for arbitrary real rate parameters.

For $m = l - 1$ independent symmetric flip coordinates, retention relative to the prepared point mass is

$$\|\mu_t - \delta_{\eta^0}\|_{\mathrm{TV}} = 1 - \left(\frac{1 + e^{-2\varphi t}}{2}\right)^m \le m\varphi t.$$

Thus $\varphi \le \varepsilon/(mT)$ is sufficient. At $l = 6$, $T = 1$, $\varepsilon = 0.05$, the displayed ceiling is 0.01 and the computed loss is 0.0485327423791. This controls retention, rather than convergence to the uniform distribution. The former smaller numerical ceiling was conservative, but did not follow by the claimed inversion of a mixing bound.

5 Fresh execution and sensitivity controls

The checker uses explicit exceptions; its scientific guards remain active under `python3 -O`. The driver performs fresh normal and optimized runs, compares both the JSON scientific report and the macro file byte for byte, and tests five deliberate defects in each mode: a changed stock ratio, broken involution, stale entropy rounding, incorrect stacking count, and an extra desorption waiting term in the cap formula. All ten defect runs are rejected. These controls test sensitivity to those defects; they are not exhaustive fault detection.

The executable reports 153 successful checks in each mode. Of these, 71 are finite mathematical or numerical checks and 82 only ensure that macro names are not duplicated. Emission of a macro is not an independent proof of its associated theorem.

6 Recovery and execution

6.1 From this TeX source alone

Copy `Article90_Verification_Code_Overleaf_LaTeX.tex` into an empty writable directory. Compile with pdfLaTeX twice, or use Overleaf's pdfLaTeX compiler. Compilation writes the embedded runtime and reports; no shell escape or Python execution is required to build the PDF. Then run:

```
python3 run_verification.py
```

The fresh results are `verification_report.json`, `verification_values_a90.tex` and `execution_audit.json`.

6.2 From this digital PDF alone

Save the embedded files with a PDF attachment viewer or Poppler:

```
pdfdetach -saveall Article90_Verification_Code_Overleaf_LaTeX.pdf
python3 run_verification.py
```

A printed or rasterized PDF does not preserve these attachments. The recorded reports are provenance; rerunning the driver creates fresh execution evidence. Python 3.9 or newer is sufficient; this release was executed with Python 3.12.14.

7 Complete verification program

verify_a90.py

```
#!/usr/bin/env python3
"""
verify_a90.py
Verification script for Article 90: "The Source Duality" (Revised per Referee Audit)
Annals of the Chrysene Formalism, Vol. 1, No. 4 (November 2026).

Uses Python standard library only (fractions.Fraction, itertools, math).
Audited finite checks for Blocks Y1--Y9; not a proof of all article claims.
Writes verification_values_a90.tex and verification_report.json.
"""

from fractions import Fraction
import itertools
import math
import json

checks = 0
def check(condition, message="verification condition failed"):
    global checks
    checks += 1
    if not condition:
        raise AssertionError(message)

def sci(value, digits=2):
    mantissa, exponent = f"{float(value):.{digits}e}".split("e")
    return mantissa + " \\times 10^{" + str(int(exponent)) + "}"

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

# ======================================================================
# BLOCK Y1: Baseline Reproduction (Articles 79 & 84) & Orthogonality
# ======================================================================
q = Fraction(10, 1)
e = [Fraction(1, 100) * (q ** n) for n in range(5)]
R_D, R_T, R_I = Fraction(42354, 1), Fraction(208, 1), Fraction(1, 1)
rho_star = R_D / R_T  # 21177/104 = 203.625

k1 = k2 = k4 = Fraction(100, 1)
x_base = e[3] / k4  # 1/10
k3 = e[2] * (R_T / R_D) * (1 + x_base)
check(k3 == Fraction(572, 105885), 'source check at line 44')

k = [None, k1, k2, k3, k4, Fraction(0, 1)]
lam = [e[n] + k[n + 1] for n in range(5)]

p = []
for n in range(5):
    prod = Fraction(1, 1)
    for j in range(n):
        prod *= k[j + 1] / lam[j]
    p.append((e[n] / lam[n]) * prod)

check(sum(p) == Fraction(1, 1), 'source check at line 56')
check(p[2] / p[4] == rho_star, 'source check at line 57')
check(p[4] / p[3] == k4 / e[3], 'source check at line 58')

Phi_min = R_D / p[2]
check(R_I <= Phi_min * p[3], 'source check at line 61')
eta_DT = (R_D + R_T) / Phi_min
eta_total = (R_D + R_I + R_T) / Phi_min
```

```
adm_weight = p[2] + p[3] + p[4]
waste_weight = p[0] + p[1]

emit("valRD", "42,354")
emit("valRT", "208")
emit("valRI", "1")
emit("valRhoStarExact", f"{rho_star.numerator}/{rho_star.denominator}")
emit("valRhoStarDec", f"{float(rho_star):.3f}")
emit("valKThreeSci", "5.40 \\times 10^{-3}")
emit("valPTwo", f"{float(p[2]):.3f}")
emit("valPFour", "4.88 \\times 10^{-3}")
emit("valPThree", "4.88 \\times 10^{-4}")
emit("valPOne", "9.99 \\times 10^{-4}")
emit("valPZero", "1.00 \\times 10^{-4}")
emit("valPhiMin", f"{round(float(Phi_min)):,}")
emit("valEtaDT", f"{float(eta_DT):.6f}")
emit("valEtaTotal", f"{float(eta_total):.6f}")
emit("valAdmWeight", f"{float(adm_weight):.6f}")
emit("valWasteWeight", "1.10 \\times 10^{-3}")

# ======================================================================
# BLOCK Y2: Mixed Arm Words, Cap Contacts & Coincidences under H_v
# ======================================================================
LOCANTS = {'D': (6, 12), 'I': (3, 6, 12), 'T': (3, 6, 9, 12)}
IOTA = {3: 9, 9: 3, 6: 12, 12: 6}

def iota_pat(pat):
    return frozenset((IOTA[loc], arm) for (loc, arm) in pat)

def enum_classes(m):
    out = {'D': [], 'I': [], 'T': []}
    for fam, locs in LOCANTS.items():
        seen = set()
        for w in itertools.product(range(m), repeat=len(locs)):
            pat = frozenset(zip(locs, w))
            pf = iota_pat(pat)
            canon = min(tuple(sorted(pat)), tuple(sorted(pf))) if fam in ('D', 'T') else tuple(sorted(pat
 ))
            if canon not in seen:
                seen.add(canon)
                out[fam].append((pat, pf, pat == pf))
    return out

c2, c3 = enum_classes(2), enum_classes(3)
check(sum(len(v) for v in c2.values()) == 21, 'source check at line 107')
check(sum(len(v) for v in c3.values()) == 78, 'source check at line 108')

blind_D = sum(1 for _, _, b in c2['D'] if b)
vis_D = sum(1 for _, _, b in c2['D'] if not b)
blind_I = sum(1 for _, _, b in c2['I'] if b)
blind_T = sum(1 for _, _, b in c2['T'] if b)
vis_T = sum(1 for _, _, b in c2['T'] if not b)
check((blind_D, vis_D, blind_I, blind_T, vis_T) == (2, 1, 0, 4, 6), 'source check at line 115')

# Specific pattern checks (0 = 'h', 1 = 'e')
pat_Dee = frozenset([(6, 1), (12, 1)])
pat_Deh = frozenset([(6, 1), (12, 0)])
pat_Deh_f = iota_pat(pat_Deh)
pat_Ieee = frozenset([(3, 1), (6, 1), (12, 1)])
pat_Ieee_f = iota_pat(pat_Ieee)
pat_Teeee = frozenset([(3, 1), (6, 1), (9, 1), (12, 1)])

v0_DD, v1_DD = len(pat_Deh & pat_Deh), len(pat_Deh & pat_Deh_f)
v0_DT, v1_DT = len(pat_Deh & pat_Teeee), len(pat_Deh_f & pat_Teeee)
v0_IDeh, v1_IDeh = len(pat_Ieee & pat_Deh), len(pat_Ieee & pat_Deh_f)
v0_IDee, v1_IDee = len(pat_Ieee & pat_Dee), len(pat_Ieee_f & pat_Dee)
v0_ITeeee, v1_ITeeee = len(pat_Ieee & pat_Teeee), len(pat_Ieee_f & pat_Teeee)
```

```
check((v0_DD, v1_DD) == (2, 0), 'source check at line 131')
check((v0_DT, v1_DT) == (1, 1), 'source check at line 132')
check((v0_IDeh, v1_IDeh) == (1, 1), "I(eee)--D(eh) cap contact must be blind (v0=v1=1)")
check((v0_IDee, v1_IDee) == (2, 2), "I(eee)--D(ee) cap contact has v0=v1=2")
check((v0_ITeeee, v1_ITeeee) == (3, 3), "I(eee)--T(eeee) cap contact has v0=v1=3")

emit("valWordsMTwo", 21)
emit("valWordsMThree", 78)
emit("valBlindDMTwo", blind_D)
emit("valVisDMTwo", vis_D)
emit("valBlindTMTwo", blind_T)
emit("valVisTMTwo", vis_T)
emit("valDeltaVDehDeh", v0_DD - v1_DD)
emit("valVDehTeeeeAligned", v0_DT)
emit("valVDehTeeeeUnaligned", v1_DT)
emit("valDeltaVDehTeeee", v0_DT - v1_DT)
emit("valVIeeeDehAligned", v0_IDeh)
emit("valVIeeeDehUnaligned", v1_IDeh)
emit("valDeltaVIeeeDeh", v0_IDeh - v1_IDeh)
emit("valVIeeeDee", v0_IDee)
emit("valVIeeeTeeee", v0_ITeeee)

# ======================================================================
# BLOCK Y3 & Y4: Cleavage Bias & Pareto Frontier
# ======================================================================
def h2(prob):
    return 0.0 if (prob <= 0.0 or prob >= 1.0) else -(prob * math.log2(prob) + (1 - prob) * math.log2(1 -
      prob))

theta_1 = 1.0 / (1.0 + math.exp(2.0))
theta_2 = 1.0 / (1.0 + math.exp(4.0))
emit("valThetaDehRgyOne", f"{theta_1:.4f}")
emit("valGainDehRgyOne", f"{1.0 - h2(theta_1):.4f}")
emit("valThetaDehRgyTwo", f"{theta_2:.4f}")
emit("valGainDehRgyTwo", f"{1.0 - h2(theta_2):.4f}")
emit("valMaxDehContactFreq", "0.2500")

def eta_arm(pi_e):
    pi_h = Fraction(1, 1) - pi_e
    return (R_D + R_T) / (R_D / (1 - pi_h ** 2) + R_T / (pi_e ** 4))

emit("valEtaArmNineNine", f"{float(eta_arm(Fraction(99, 100))):.4f}")
emit("valEtaArmNineFive", f"{float(eta_arm(Fraction(95, 100))):.4f}")
emit("valEtaArmNineZero", f"{float(eta_arm(Fraction(90, 100))):.4f}")
emit("valEtaArmSevenFive", f"{float(eta_arm(Fraction(75, 100))):.4f}")
emit("valEtaArmFiveZero", f"{float(eta_arm(Fraction(50, 100))):.4f}")
emit("valVisFreqNineNine", "3.92 \\times 10^{-4}")
emit("valVisFreqNineFive", "9.03 \\times 10^{-3}")
emit("valVisFreqNineZero", "3.24 \\times 10^{-2}")
emit("valVisFreqSevenFive", "0.1406")
emit("valVisFreqFiveZero", "0.2500")

# ======================================================================
# BLOCK Y5 & Y6: Initiator Doping & Hexamer Cost Partition
# ======================================================================
def coop(x_val):
    return q * rho_star / (x_val * (1 + x_val))

emit("valCoopXPointOne", f"{round(float(coop(Fraction(1, 10)))):,}")
emit("valCoopXOne", f"{round(float(coop(Fraction(1, 1)))):,}")
emit("valCoopXTen", f"{float(coop(Fraction(10, 1))):.1f}")
emit("valFISqBaseline", sci((Fraction(1,10)/(rho_star+1+Fraction(1,10)))**2))
emit("valFISqXOne", sci((1/(rho_star+2))**2))
emit("valFISqXTen", sci((10/(rho_star+11))**2))

theta_II_1 = 1.0 / (1.0 + math.exp(1.0))
theta_II_2 = 1.0 / (1.0 + math.exp(2.0))
emit("valGainIIContactRgyOne", f"{1.0 - h2(theta_II_1):.4f}")
```

```
emit("valGainIIContactRgyTwo", f"{1.0 - h2(theta_II_2):.4f}")

def spec_cost(rho_v, x_v):
    fD = rho_v / (rho_v + 1 + x_v)
    fT = Fraction(1, 1) / (rho_v + 1 + x_v)
    return -math.log2(float((fD ** 3) * (fT ** 3)))

sc100 = spec_cost(Fraction(100, 1), Fraction(1, 10))
sc200 = spec_cost(Fraction(200, 1), Fraction(1, 10))
emit("valSpecCostOneHundred", f"{sc100:.1f}")
emit("valSpecCostTwoHundred", f"{sc200:.1f}")
emit("valRegCostSix", 5)
emit("valRegCostFiftyFour", 53)
emit("valRegCostBoxZ", 45)
emit("valTotalCostTwoHundred", f"{sc200 + 5.0:.1f}")
emit("valRegShareTwoHundred", round(100.0 * 5.0 / (sc200 + 5.0)))

# ======================================================================
# BLOCK Y7 & Y8: Persistence Semigroup & Connes Distance
# ======================================================================
r_s, a_s, d_s, e4_s = Fraction(1, 1000), Fraction(5, 1), Fraction(5, 100), Fraction(100, 1)
buf = 1 + a_s / d_s
check(buf == Fraction(101, 1), 'source check at line 220')
check(buf / r_s == Fraction(101000, 1), 'source check at line 221')
emit("valBufAmplification", 101)
emit("valMeanReturnTime", f"{int(1/d_s+buf/r_s):,}")
emit("valMeanReturnTimeStack", f"{int(buf/r_s):,}")
emit("valMeanReturnTimeColumn", f"{int(1/d_s+buf/r_s):,}")
emit("valResidualPiFourBeta", sci(1/(1+e4_s/r_s*buf),1))
emit("valMaxReturnRateSix", "8.33 \\times 10^{-3}")
emit("valMaxFlipRateSix", sci(Fraction(5,100)/5))
emit("valDimQOne", 4**1)
emit("valDimQSix", f"{4**6:,}")
emit("valConnesLowerSix", f"{(6-1)/2:.2f}")
emit("valConnesLowerFiftyFour", f"{(54-1)/2:.2f}")

# ======================================================================
# BLOCK Y9: 3D Interior Lattice B(3,3,6) Terminal Blindness (Target 14.10)
# ======================================================================
nx, ny, nz = 3, 3, 6
nodes_box = nx * ny * nz
bridges_box = (nx - 1) * ny * nz + nx * (ny - 1) * nz
stacks_box = nx * ny * (nz - 1)
spanning_cols = nx * ny - 1
check(stacks_box + spanning_cols == nodes_box - 1 == 53, 'source check at line 240')

total_arms_box = 4 * nodes_box
free_arms_box = total_arms_box - 2 * bridges_box
complete_sites_box = max(0, nx - 2) * max(0, ny - 2) * max(0, nz - 2)

check((nodes_box, bridges_box, stacks_box, total_arms_box, free_arms_box, complete_sites_box) == (54, 72,
      45, 216, 72, 4), 'source check at line 246')

def s_free(kx, ky):
    s = set()
    if kx == 0: s.add(3)
    if kx == nx - 1: s.add(9)
    if ky == 0: s.add(12)
    if ky == ny - 1: s.add(6)
    return frozenset(s)

layer_free = sum(len(s_free(kx, ky)) for kx in range(nx) for ky in range(ny))
check(layer_free == 12, 'source check at line 257')
check(layer_free * nz == 72, 'source check at line 258')
check(s_free(1, 1) == frozenset(), 'source check at line 259')
check(frozenset(IOTA[loc] for loc in s_free(1, 1)) == s_free(1, 1), 'source check at line 260')

emit("valBoxNodes", nodes_box)
```

```
emit("valBoxBridges", bridges_box)
emit("valBoxStacks", stacks_box)
emit("valBoxSpanningContacts", spanning_cols)
emit("valBoxTotalArms", total_arms_box)
emit("valBoxFreeArms", free_arms_box)
emit("valBoxCompleteSites", complete_sites_box)
emit("valBoxCentralContacts", nz - 1)
emit("valBoxEdgeContacts", 4 * (nz - 1))
emit("valBoxCornerContacts", 4 * (nz - 1))


# Audit additions: independent expected values, orbit probability mass,
# cap first-step equations and corrected retention example.
check(rho_star == Fraction(1629, 8), "stock ratio")
check(round(float(eta_DT), 6) == 0.998413, "D/T efficiency")
check(round(float(eta_total), 6) == 0.998437, "total efficiency")
check(round(float(adm_weight), 6) == 0.998901, "admitted mass")
check(round(float(Phi_min)) == 42630, "minimum flux rounding")
check(round(1-h2(theta_1), 4) == 0.4729, "first entropy gain")
check(round(1-h2(theta_2), 4) == 0.8700, "second entropy gain")
check(round(1-h2(theta_II_1), 4) == 0.1601, "I/I gain")
for pi, expected in [(99,0.9997),(95,0.9964),(90,0.9875),
                     (75,0.9286),(50,0.7117)]:
    check(round(float(eta_arm(Fraction(pi,100))),4)==expected,
          "arm efficiency table")
check(round(sc100,1)==20.0 and round(sc200,1)==23.0,
      "hexamer selection costs")
check(int(1/d_s+buf/r_s)==101020, "column mean includes desorption")
# Enumerate and normalize quotient probabilities under the support
# stabilizer: D,T have the full involution; fixed canonical I has only id.
for m in (2,3):
    weights=[Fraction(i+1,m*(m+1)//2) for i in range(m)]
    for family, classes in enum_classes(m).items():
        mass=Fraction(0)
        for pat, flipped, blind in classes:
            prob=math.prod(weights[arm] for loc,arm in pat)
            mass += prob if (family=='I' or blind) else 2*prob
        check(mass==1, f"normalized {family} quotient at m={m}")
# Finite boundary-terminal checks are separate from chemical realizability.
for ix in range(nx):
    for iy in range(ny):
        free=s_free(ix,iy)
        reflected=frozenset(IOTA[loc] for loc in free)
        check(reflected==s_free(nx-1-ix,ny-1-iy),
              "reflected free-terminal support")
        check((reflected==free)==((ix,iy)==(1,1)),
              "only the central free-terminal pattern is fixed")
check((len(pat_Ieee & pat_Ieee),
       len(pat_Ieee & pat_Ieee_f))==(3,2), "I/I coincidence")
# Six transient states: C3,S3,B3,B4,S4,C4(absorbing).
# Solve first-step equations directly over the rationals.
def solve_linear(matrix, rhs):
    a=[list(row)+[value] for row,value in zip(matrix,rhs)]
    n=len(a)
    for col in range(n):
        pivot=next(i for i in range(col,n) if a[i][col])
        a[col],a[pivot]=a[pivot],a[col]
        scale=a[col][col]
        a[col]=[value/scale for value in a[col]]
        for row in range(n):
            if row!=col:
                scale=a[row][col]
                a[row]=[x-scale*y for x,y in zip(a[row],a[col])]
    return [row[-1] for row in a]

def cap_mean(d,a,r,e,k,e4,a4,r4):
    z=Fraction(0)
    # -Q times mean = 1, for transient states C3,S3,B3,B4,S4.
```

```
    matrix=[[d,-d,z,z,z],[-a,a+r,-r,z,z],
            [z,-e,e+k,-k,z],[z,z,z,e4,-e4],
            [z,z,z,-r4,r4+a4]]
    means=solve_linear(matrix,[Fraction(1)]*5)
    formula=1/d+(1+a/d)/r*(1+e/k)+1/k+1/e4+(1+r4/e4)/a4
    check(means[0]==formula,"cap first-step mean identity")
    return formula
cap_example=cap_mean(d_s,a_s,r_s,Fraction(10),Fraction(100),
                     Fraction(100),Fraction(5),Fraction(1,1000))
cap_mean(*map(Fraction,[2,3,5,7,11,13,17,19]))
# Retention is distance to the prepared point mass, not to uniform.
phi=0.01; time=1.0; edges=5; epsilon=0.05
retention_tv=1-((1+math.exp(-2*phi*time))/2)**edges
check(retention_tv<=epsilon,"corrected sufficient flip ceiling")
check(abs(retention_tv-0.04853274237910499)<1e-7,"retention probability")
for rate in (0.0,0.0005,0.01,0.1,1.0):
    loss=1-((1+math.exp(-2*rate))/2)**5
    check(loss<=5*rate+1e-14,"finite retention bound samples")
# These are numbers entering a conditional witness bound, not a check
# of a Dirac commutator or a general spectral metric theorem.
check(4**6==4096 and (6-1)/2==2.5 and (54-1)/2==26.5,
      "quotient dimensions and conditional witness arithmetic")
report={"article":90,"volume":1,"issue":4,"date":"November 2026",
        "checks_passed":checks,"macros":len(macros),
        "mathematical_checks":checks-len(macros),
        "macro_uniqueness_checks":len(macros),
        "scope":"finite arithmetic, enumerations and selected numerical examples",
        "entropy_gains":[1-h2(theta_1),1-h2(theta_2)],
        "arm_efficiencies_075_050":[float(eta_arm(Fraction(3,4))),
                                    float(eta_arm(Fraction(1,2)))],
        "mean_return_stack":str(buf/r_s),
        "mean_return_column":str(1/d_s+buf/r_s),
        "cap_first_step_example":str(cap_example),
        "flip_ceiling_sufficient":phi,
        "retention_tv_at_ceiling":retention_tv}
with open("verification_report.json","w",encoding="utf-8") as f:
    json.dump(report,f,indent=2,sort_keys=True); f.write("\n")

with open("verification_values_a90.tex", "w", encoding="utf-8") as f:
    f.write("% Generated automatically by verify_a90.py (Blocks Y1--Y9)\n")
    for k_name, v_str in macros.items():
        f.write(f"\\newcommand{{\\{k_name}}}{{{v_str}}}\n")

print(f"VERIFICATION COMPLETE: {checks} checks passed; {len(macros)} macros written.")
```

8 Complete execution audit driver

run_verification.py

```
#!/usr/bin/env python3
"""Run Article 90 in normal/-O modes and reject five deliberate defects."""
import hashlib, json, subprocess, sys, tempfile
from pathlib import Path
root = Path(__file__).resolve().parent
source = root / 'verify_a90.py'
text = source.read_text(encoding='utf-8')
mutations = {
    'stock_ratio': ('Fraction(42354, 1)', 'Fraction(42355, 1)'),
    'involution': ('3: 9, 9: 3, 6: 12, 12: 6',
                   '3: 9, 9: 3, 6: 6, 12: 12'),
    'entropy_rounding': ('== 0.4729, "first entropy gain"',
                         '== 0.4728, "first entropy gain"'),
    'stack_count': ('stacks_box = nx * ny * (nz - 1)',
                    'stacks_box = nx * ny * nz'),
    'cap_waiting_time': ('formula=1/d+(1+a/d)/r',
                         'formula=2/d+(1+a/d)/r'),
}
outputs = {}
rejected = []
with tempfile.TemporaryDirectory(prefix='article90-verify-') as temp:
    temp = Path(temp)
    for optimized in (False, True):
        mode = 'optimized' if optimized else 'normal'
        folder = temp / mode
        folder.mkdir()
        command = [sys.executable] + (['-O'] if optimized else [])
        run = subprocess.run(command + [str(source)], cwd=folder,
                             capture_output=True, text=True)
        if run.returncode:
            raise RuntimeError(run.stdout + run.stderr)
        outputs[mode] = {name: (folder/name).read_bytes() for name in
                         ('verification_report.json',
                          'verification_values_a90.tex')}
        for name, (old, new) in mutations.items():
            if text.count(old) != 1:
                raise RuntimeError('Mutation target is not unique: ' + name)
            bad = folder / (name + '.py')
            bad.write_text(text.replace(old, new), encoding='utf-8')
            test = subprocess.run(command + [str(bad)], cwd=folder,
                                  capture_output=True, text=True)
            if test.returncode == 0 or 'AssertionError' not in test.stderr:
                raise RuntimeError('Negative control not rejected: '+name)
            rejected.append(mode + ':' + name)
    if outputs['normal'] != outputs['optimized']:
        raise RuntimeError('Normal and optimized scientific outputs differ')
    for name, data in outputs['normal'].items():
        (root/name).write_bytes(data)
report = {
    'source_sha256': hashlib.sha256(source.read_bytes()).hexdigest(),
    'python_version': sys.version.split()[0],
    'normal_and_optimized_outputs_identical': True,
    'negative_controls': rejected,
    'mathematical_checks_per_run': 71,
    'macro_uniqueness_checks_per_run': 82,
    'checks_per_run': 153,
}
(root/'execution_audit.json').write_text(
    json.dumps(report, indent=2, sort_keys=True)+'\n', encoding='utf-8')
print('PASS: 153 checks per mode (71 mathematical, 82 macro uniqueness).')
print('PASS: normal/-O outputs identical; 10 of 10 defect runs rejected.')
```