ANNALS OF THE CHRYSENE FORMALISM

RESEARCH ARTICLE | AI-AUGMENTED PEER-REVIEWED | VOL. 1, NO. 5 | NOVEMBER 2026

Verification Code for Article 103

From Fibrated Chrysene Networks to Molecular Machines

Registry-Programmed Geometry, Electromechanical Realization, and Constructive Existence

Charles D. Schaper, Ph.D.

Annals of the Chrysene Formalism, Volume 1, Number 5

Standalone computational supplement, November 2026

Abstract

This supplement supplies executable, deterministic checks for the explicit finite model in Article 103. Exact rational arithmetic checks finite geometry, bridge identities, energy completion, exponential-series bounds, control-path budgets, and checkpoint probability arithmetic. Floating-point calculations independently evaluate the occupancy law, reversible transition generator, spectral mixing bounds, and finite perturbation examples. Negative controls expose lost selectivity, absent control coupling, singular stiffness, inadmissible geometry, omitted filter multiplicity, and nonunique historical decoding. The program is reproduced in full and needs no manuscript source files at runtime. These checks are neither a proof of every analytic theorem nor empirical validation of chrysene chemistry, self-assembly, or molecular manufacturing.

1 Scope and interpretation

The principal witness uses sixteen indexed cores, two rigid blocks, aperture $a \in [2, 6]$, stiffness $K = 16$, and contact scores $E_T = 10 + 6r$, $E_C = 10$. It declares a finite minimized-basin law, not the continuous Gibbs integral over the aperture. The numerical kinetic completion is an explicitly selected reversible three-state generator; its attempt frequency is not deduced from the binding energies.

The executable distinguishes four levels of evidence:

1. **Exact finite identities and arithmetic certificates.** Python's `Fraction` represents rational inputs without roundoff. These include the positive partial sums used by the manuscript's analytic exponential-series argument. Python execution is not a proof-assistant kernel check.

2. **Exact finite samples.** A rational calculation at specified sample points is exact at those points, but is not a quantified proof on an interval. Piecewise-affine slope and margin budgets are separately checked; the manuscript proves the continuous statements.

3. **Floating-point checks.** NumPy and SciPy evaluate matrices, exponentials, and eigenvalues. The tolerances are absolute and relative 2×10^{-11}. These are not interval enclosures or certified eigensolver bounds.

4. **Finite robustness examples.** An explicitly enumerated $3^4 = 81$ parameter grid checks the stated thresholds. This grid does not establish the open-neighborhood theorem, arbitrary perturbations, or chemical feasibility.

The check count includes parameterized cases, not distinct mathematical theorems. No random experiment is used. The script exits with a nonzero status if any check fails and records every check in the JSON output.

2 Coverage map and exclusions

The references below use named results rather than renumberable theorem numbers.

Finite restriction; port covariance.
Address counts, retained typed edge counts, an injective illustrative four-port embedding, ordered endpoint closure, and proper rigid-motion covariance are checked. The chosen four-port coordinates are a rational test fixture, not an atomistic chrysene structure. The fixture also verifies that its ports lie inside the conservative unit envelopes, so an incident attachment sleeve is necessary before exterior routing. Radius-5/4 collars have separation budget $4 - 2(5/4) = 3/2$ and reference corridor margin $9/4$. Boundary termination chemistry and general inheritance results are not certified by this fixture.

Citation: Schaper, C. D. (2026). From Fibrated Chrysene Networks to Molecular Machines: Registry-Programmed Geometry, Electromechanical Realization, and Constructive Existence. *Annals of the Chrysene Formalism*, 1(5):1975–1990.
https://chrysene.com/index.php/acf/article/view/103
© 2026 Charles D. Schaper. Distributed under the CC Attribution-NonCommercial-NoDerivatives 4.0. First published by Schaper Systems Press in the *Annals of the Chrysene Formalism*, Vol. 1, No. 5 (2026). ISSN 3144-2021.

Core clearance and substrate corridor.
The reference determinant, obliquity, squared center separations, half-space clearance arithmetic, and reference-to-moving coordinate map are checked. The route map H_a is evaluated at knots, ports, and finite secants, and its uniform lower slope and tube-radius budget are checked. The program does not construct chemically valid tethers or prove the topological existence of all exterior routes. Exposed exit anchors/sleeves and nonincident reference-route margins remain declared realization data.

Stable branch; capture–release witness.
Square completion, interior minimizers, endpoint free energies, relaxation samples, exact exponential-series partial sums and rational probability bounds are tested. Odds, registry discrimination and the reported numerical probabilities are evaluated independently.

Local flow and energy accounting.
Chain-rule identities are checked at exact rational states. A separate occupancy jump and fixed-aperture control reset telescope the energy correctly. In the illustrated binding event the jump changes energy by -4 and the reset contributes work $+1/2$; neither is silently counted as free mechanical motion.

Reversible kinetics and controlled operation.
Generator row sums, detailed balance, stationarity, residence times, symmetric similarity, Dirichlet form, positive spectral gap and the mixing inequality for all three point-mass initial laws are numerically checked for three parameter choices. Convexity of total variation extends a point-mass check at a fixed time to initial mixtures, but the finitely many checked times do not prove the continuous-time theorem.

Active clock and path lift.
The frozen-clock semigroup, exit-rate upper bound 2ν, exact joint-success arithmetic and smoothstep path budgets are checked. Availability is 0 or 1. The example with active dwell $T = 10$, $\nu = 1$, all paused interface durations bounded by $\tau_{\max} = 2$, and fixed overhead 4 has expected-time upper bound 54. This is conditional arithmetic under the manuscript's resource, sensing, interface and timing assumptions, not a measured molecular duration. Success is a probabilistic checkpoint event, not a guarantee on every trajectory.

Symmetry transport.
Matrix permutation verifies invariance under a transported generator. The $r = 0$ negative control separately removes physical target discrimination; it is not the same operation as merely renaming states. A further negative control uses the same active generator and initial occupancy with two different initial pause durations: the physical-time laws differ. Physical-time equivalence therefore requires matching pause and reset rules, not merely equal active-time generators.

Energy-gap filtering; finite coding section.
Weighted finite filtering, a sharp case, the empty rejected set, a coding section and its nonunique historical inverse are checked. A resource-accounted toy extension illustrates interface compatibility, not a chemical production process. Two negative controls keep the shared code/product coordinates identical while withholding a fresh packet or a catalyst required by the full guard; both expose a blocked next transition. Shared-interface equality alone cannot certify these missing prerequisites.

Retention; residual and sampled-path certificates.
Perturbation and Hessian-error budgets are checked, together with a quadratic residual example and a Lipschitz path budget. Negative examples demonstrate the necessity of container membership, a neighborhood Hessian bound, and control of between-sample behavior. The script does not establish general compactness, smoothness, Lipschitz constants or admissibility hypotheses from atomistic data.

General equilibrium existence, the implicit-function theorem, persistence under all sufficiently small perturbations, continuum topology, infinite-dimensional fibrations, and chemical transitions remain the domain of the manuscript's arguments and future realization data. The separate Lean supplement has its own explicitly stated coverage.

3 Reproduction and recorded results

The recorded run used Python 3.12.14, NumPy 2.3.5 and SciPy 1.17.0. Install the two pinned runtime dependencies in an appropriate Python environment. Save the full source below as `verify_article103.py`, then run:

```
python -m pip install numpy==2.3.5 scipy==1.17.0
python verify_article103.py --json results_article103.json
```

The delivered source, `requirements.txt`, JSON and console transcript accompany this document. No network access, manuscript files or external data are needed after dependency installation. JSON includes versions, tolerances, source SHA-256, group counts, numerical metrics and all individual check outcomes. Re-running with other library versions may change the last floating-point digits.

The recorded run passed **472 of 472 checks**, with zero failures. The target probability at capture is 0.9865944031662418; the empty probability at release is 0.9706172463778026. The minimum values on the specified 81-point perturbation grid are approximately 0.9865455583 and 0.9704438513, respectively. Exact checkpoint multiplication gives

$$(0.98 - 10^{-4})(0.97 - 10^{-4}) = 0.95040501 > 0.95.$$

The stricter numerical dwell target 10^{-5} is used only in the spectral test, leaving margin relative to the theorem's 10^{-4} tolerance.

Group	Checks	Group	Checks
Geometry	25	Routing	25
Mechanics	73	Work	19
Probability	28	Robustness grid	81
Kinetics	24	Active clock	8
Paths	161	Symmetry	3
Extensions	8	Certificates	3
Negative controls	14	Total	472

Source SHA-256 (for the delivered raw Python file):

```
dd69e6479135a9fbbd79441678f19c4a0da8781a1f17dab92c79accc588da1b1
```

4 Complete executable source

Long source lines wrap typographically in this document. The raw `.py` companion is the canonical executable file. The following listing embeds the complete source directly, so compiling this LaTeX document requires neither a shell escape nor a separate source-code input.

```
#!/usr/bin/env python3
"""Deterministic finite-model checks for Article 103, not chemical validation.
Run: python verify_article103.py --json results_article103.json
Exact certificates use Fraction. Floating-point checks use ATOL below.
No simulated experiments, random sampling, or unreported solver tolerances.
"""
import argparse
from collections import Counter
from fractions import Fraction as F
import hashlib
import itertools as it
import json
import math
from pathlib import Path
import platform
import sys

import numpy as np
import scipy
from scipy.linalg import expm

```

```
ATOL = 2e-11
checks = []
metrics = {}


def check(group, name, condition, kind="exact"):
    checks.append(dict(group=group, name=name, kind=kind,
                       passed=bool(condition)))


def close(x, y):
    return np.allclose(x, y, atol=ATOL, rtol=ATOL)


def sqdist(x, y):
    return sum((a-b)**2 for a, b in zip(x, y))


def add(x, y):
    return tuple(a+b for a, b in zip(x, y))


def sub(x, y):
    return tuple(a-b for a, b in zip(x, y))


def scale(t, x):
    return tuple(t*a for a in x)


addresses = list(it.product((0, 1, 3, 4), (0, 1), (0, 1)))


def position(k, a):
    i, j, z = k
    return (4*(i if i < 2 else i-3)+j
            + (0 if i < 2 else a+7), 4*j, 4*z)


def h(x, a):
    if x <= 6:
        return x
    if x <= 11:
        return 6+a*(x-6)/5
    return x+a-5


# Geometry: finite identities, then finite samples of continuous formulas.
ports = {3: (F(-1, 2), 0, 0), 9: (F(1, 2), 0, 0),
         12: (0, F(-1, 2), 0), 6: (0, F(1, 2), 0)}
edges = {d: [(u, v) for u in addresses for v in addresses
             if tuple(v[j]-u[j] for j in range(3))
             == tuple(int(j == d) for j in range(3))]
         for d in range(3)}
check("geometry", "sixteen distinct addresses", len(set(addresses)) == 16)
check("geometry", "edge counts x8 y8 z8", [len(edges[d]) for d in range(3)] == [8, 8, 8])
check("geometry", "reference determinant 64", 4*4*4 == 64)
check("geometry", "oblique generators", sum(x*y for x, y in zip((4, 0, 0), (1, 4, 0))) == 4)
check("geometry", "reference address embedding", all(position(k, F(5)) == (4*k[0]+k[1], 4*k[1], 4*k[2]) for
```

```
     k in addresses))
# Ports in this fixture lie inside conservative unit envelopes.
# Incident attachment sleeves are necessary before exterior routing begins.
check("geometry", "interior port needs incident attachment sleeve",
      all(sqdist(p, (0, 0, 0)) == F(1, 4) < 1
          for p in ports.values()))
check("geometry", "disjoint radius-five-fourths collar gap",
      F(4)-2*F(5, 4) == F(3, 2) > 0)
check("geometry", "reference collar corridor halfspace margin",
      F(17, 2)-F(5)-F(5, 4) == F(9, 4))
sites = [add(position(k, F(5)), p) for k in addresses for p in ports.values()]
check("geometry", "illustrative four-port embedding injective", len(set(sites)) == len(sites))
# Explicit reference bridge vectors: endpoint difference closes each bridge.
for d, outport, inport, vector in [(0, 9, 3, (3, 0, 0)), (1, 6, 12, (1, 3, 0))]:
    check("geometry", f"ordered port bridge closure axis {d}", all(
        sub(add(position(v, F(5)), ports[inport]),
            add(position(u, F(5)), ports[outport])) == vector
        for u, v in edges[d]))
# Illustrative coordinates are not asserted to be actual chrysene atom positions.
Q = lambda p: (-p[1], p[0], p[2])
translation = (2, -3, 5)
check("geometry", "proper rotation preserves distances", all(
    sqdist(Q(x), Q(y)) == sqdist(x, y) for x, y in it.combinations(ports.values(), 2)))
check("geometry", "port covariance including translations", all(
    add(add(Q(position(k, F(5))), translation), Q(p))
    == add(Q(add(position(k, F(5)), p)), translation)
    for k in addresses for p in ports.values()))
for a in (F(2), F(11, 4), F(3), F(19, 4), F(5), F(6)):
    points = [position(k, a) for k in addresses]
    check("geometry", f"squared core separation a={a}", min(sqdist(x, y) for x, y in it.combinations(
    points, 2)) >= 16)
    check("geometry", f"corridor halfspace bound a={a}", a/2-F(1, 4) >= F(3, 4))
    check("routing", f"H fixes/translates ports a={a}", all(
        h(add(position(k, F(5)), p)[0], a)
        == add(position(k, a), p)[0]
        for k in addresses for p in ports.values()))
    check("routing", f"H knots and centerline a={a}", h(F(6), a) == 6 and h(F(11), a) == 6+a and h(F(17, 2)
    , a) == 6+a/2)
    check("routing", f"positive slopes bound a={a}", min(F(1), a/5) >= F(2, 5))
    grid = [F(x, 2) for x in range(-2, 29)]
    check("routing", f"sampled secant lower bound a={a}", all(abs(h(x, a)-h(y, a)) >= F(2, 5)*abs(x-y) for
     x, y in it.combinations(grid, 2)), "exact finite samples")
check("routing", "uniform radius arithmetic", F(2, 5)-F(1, 4)-F(3, 40) == F(3, 40))
check("negative", "a=0 loses corridor margin", F(0)/2-F(1, 4) < 0)


def energy(a, c, r, s, K=F(16), b=None, contact=None):
    b = (F(0) if s == 0 else F(4)) if b is None else b
    E = (F(0), F(10+6*r), F(10))[s] if contact is None else contact
    return K/2*(a-c)**2+b*a-E


def basin(c, r):
    return np.array([0., 4*c-10-6*r-.5, 4*c-10-.5])


def law(c, r, beta=1., activity=(1., 1.)):
    w = np.exp(-beta*basin(c, r))*np.array([1., *activity])
    return w/w.sum()
```

```


for c in (F(3), F(7, 2), F(4), F(9, 2), F(5)):
    for r, s in it.product((0, 1), range(3)):
        b = F(0 if s == 0 else 4)
        amin = c-b/16
        free = b*c-(0, 10+6*r, 10)[s]-b*b/32
        check("mechanics", f"minimizer interior c={c} r={r} s={s}", F(11, 4) <= amin <= 5 and 16*(amin-c)+
    b == 0)
        check("mechanics", f"square completion c={c} r={r} s={s}", all(energy(a, c, r, s) == 8*(a-amin)**2+
    free for a in (F(2), F(3), F(9, 2), F(6))))
check("mechanics", "energy endpoint values", close(basin(3, 1), [0, -4.5, 1.5]) and close(basin(5, 1), [0,
     3.5, 9.5]), "floating point")
for a0, c, s in it.product((2., 6.), (3., 5.), range(3)):
    amin = c-(0, .25, .25)[s]
    trajectory = amin+(a0-amin)*np.exp(-16*np.linspace(0, 2, 101))
    check("mechanics", f"relaxation samples {a0,c,s}", np.all((trajectory >= 2) & (trajectory <= 6)) and
    abs(trajectory[-1]-amin) < 1e-12, "floating point samples")
# An exact work-identity substitution at nonstationary states.
for a, c, dc, b in it.product((F(3), F(5)), (F(3), F(4)), (F(-1), F(2)), (F(0), F(4))):
    g = 16*(a-c)+b
    da = -F(2)*g
    check("work", f"chain rule {a,c,dc,b}", g*da-16*(a-c)*dc == -2*g*g-16*(a-c)*dc)
# Occupancy changes at fixed a; a control reset then has separate work.
# Capture empty equilibrium a=3,c=3; binding to T, then c=13/4.
a, cminus, cplus = F(3), F(3), F(13, 4)
U_before = energy(a, cminus, 1, 0)
U_jump = energy(a, cminus, 1, 1)
U_reset = energy(a, cplus, 1, 1)
reservoir_jump = U_jump-U_before
reset_work = U_reset-U_jump
check("work", "nonzero fixed-aperture reset cost", reset_work == F(1, 2))
check("work", "occupancy and control energy telescope",
      U_reset-U_before == reservoir_jump+reset_work)
check("work", "reset gives zero-velocity lift endpoint",
      16*(a-cplus)+4 == 0)
metrics["reset_example"] = dict(reservoir_jump=float(reservoir_jump),
                                reset_work=float(reset_work))
check("negative", "zero transverse stiffness not positive definite", np.linalg.eigvalsh(np.diag([16., 0.]))
    .min() <= 0, "floating point")


def partial_exp(x, n):
    return sum((x**k/F(math.factorial(k)) for k in range(n+1)), F(0))


for x, n, bound in [(F(9, 2), 7, 80), (F(3, 2), 3, 4), (F(7, 2), 9, 33), (F(19, 2), 11, 10000)]:
    check("probability", f"positive exp series x={x} n={n}", partial_exp(x, n) > bound)
check("probability", "capture rational certificate", F(80)/(80+1+F(1, 4)) == F(64, 65) and F(64, 65) > F
    (98, 100))
check("probability", "release rational certificate", F(1, 33)+F(1, 10000) < F(3, 97))
metrics["capture_law"] = law(3, 1).tolist()
metrics["release_law"] = law(5, 1).tolist()
check("probability", "numerical threshold margins", law(3, 1)[1] > .98 and law(5, 1)[0] > .97, "floating
    point")
for c in np.linspace(3, 5, 21):
    p, p0 = law(c, 1), law(c, 0)
    check("probability", f"odds and registry contrast c={c:.2f}", close(p[1]/p[2], math.exp(6)) and close(
    p0[1], p0[2]) and p[1] > p0[1], "floating point samples")
```

```
check("negative", "r0 destroys target selectivity", law(3, 0)[1] < .5, "floating point")
def no_aperture_binding_law(c):
    # Setting all b_s=0 makes minimized energies independent of rest aperture.
    minimized = [float(energy(F(c), F(c), 1, s, b=F(0))) for s in range(3)]
    weights = np.exp(-np.array(minimized))
    return weights/weights.sum()


check("negative", "no aperture binding term removes c dependence",
      close(no_aperture_binding_law(3), no_aperture_binding_law(5))
      and no_aperture_binding_law(5)[0] < .97, "floating point")
# A finite grid is evidence of examples, not a proof of the open-family theorem.
robust_min = [1., 1.]
for K, b, ET, EC in it.product((15.99, 16., 16.01), (3.999, 4., 4.001), (15.999, 16., 16.001), (9.999, 10., 10.001)):
    good = True
    for j, c in enumerate((3., 5.)):
        f = np.array([0., b*c-ET-b*b/(2*K), b*c-EC-b*b/(2*K)])
        p = np.exp(-f); p /= p.sum()
        value = p[1 if j == 0 else 0]
        robust_min[j] = min(robust_min[j], value)
        good &= K > 0 and 2 < c-b/K < 6 and value > (.98 if j == 0 else .97)
    check("robustness", f"finite parameter grid {K,b,ET,EC}", good, "floating point samples")
metrics["finite_grid_min_capture_release"] = robust_min


def kinetics(c, r=1, nu=1., beta=1., activity=(1., 1.)):
    pi = law(c, r, beta, activity)
    w = pi[1:]/pi[0]
    Qm = np.zeros((3, 3))
    Qm[0, 1:] = nu*w/(1+w)
    Qm[1:, 0] = nu/(1+w)
    Qm -= np.diag(Qm.sum(axis=1))
    return pi, Qm, w


for c, r, nu, beta, activity in [(3., 1, 1., 1., (1., 1.)), (5., 1, 1., 1., (1., 1.)), (4., 0, .7, 1.2, (.3, 2.))]:
    pi, qm, w = kinetics(c, r, nu, beta, activity)
    tag = f"c={c} r={r}"
    flux = pi[:, None]*qm
    check("kinetics", "generator and balance "+tag, close(qm.sum(axis=1), 0) and close(pi@qm, 0) and close(flux, flux.T) and np.all(qm[~np.eye(3, dtype=bool)] >= 0), "floating point")
    check("kinetics", "residence and rate ratios "+tag, close(-1/np.diag(qm)[1:], (1+w)/nu) and close(qm[0, 1:]/qm[1:, 0], w), "floating point")
    sym = -np.sqrt(pi)[:, None]*qm/np.sqrt(pi)[None, :]
    eig = np.linalg.eigvalsh(sym)
    gap = eig[1]
    check("kinetics", "reversible positive spectral gap "+tag, close(sym, sym.T) and abs(eig[0]) < ATOL and gap > 0, "floating point")
    f = np.array([2., -1., 3.])
    dirichlet = .5*sum(pi[i]*qm[i,j]*(f[j]-f[i])**2 for i in range(3) for j in range(3))
    check("kinetics", "Dirichlet identity "+tag, close(-np.dot(pi*f, qm@f), dirichlet), "floating point")
    prefactor = .5*math.sqrt(1/pi.min()-1)
    dwell = math.log(prefactor/1e-5)/gap
    metrics["kinetics_"+tag] = dict(gap=float(gap), dwell_for_bound_1e_5=dwell)
    for t in (0., .5, 2., dwell):
        transition = expm(t*qm)
        bound = .5*np.exp(-gap*t)*np.sqrt(1/pi-1)
```

```
        tv = .5*np.abs(transition-pi).sum(axis=1)
        check("kinetics", f"all point-mass mixing {tag} t={t:.4g}", np.all(tv <= bound+ATOL) and close(transition.sum(axis=1), 1), "floating point")
    check("clock", "exit rate domination "+tag, max(-np.diag(qm)) <= 2*nu, "floating point")
    check("clock", "paused clock semigroup "+tag, close(expm(.7*qm)@expm(0*qm)@expm(1.3*qm), expm(2*qm)), "floating point")
    perm = np.array([[1,0,0], [0,0,1], [0,1,0]])
    check("symmetry", "transported generator and observable "+tag, close(expm(2*(perm@qm@perm.T)), perm@expm(2*qm)@perm.T), "floating point")
# Identical active Q does not identify physical-time laws with different pauses.
pi_pause, Q_pause, _ = kinetics(3.)
p_initial = np.array([1., 0., 0.])
physical_t = 1.
no_pause = p_initial@expm(physical_t*Q_pause)
initial_pause = p_initial@expm(max(0., physical_t-2.)*Q_pause)
check("negative", "same active generator different physical clock laws",
      close(initial_pause, p_initial) and not close(no_pause, initial_pause),
      "floating point")
check("clock", "joint checkpoint success certificate", F(9799, 10000)*F(9699, 10000) == F(95040501, 100000000) and F(95040501, 100000000) > F(95, 100))
# Smoothstep chi has max chi'=3/2; tau=2 gives correction <=27/256.
check("paths", "uniform lift control correction", F(9, 4)*F(3, 2)/(16*2) == F(27, 256) and F(27, 256) < F(1, 8))
for left, right in it.product((F(11,4), F(3), F(19,4), F(5)), repeat=2):
    for b in (F(0), F(4)):
        for u in (F(0), F(1,4), F(1,2), F(3,4), F(1)):
            a = left+(right-left)*(3*u*u-2*u*u*u)
            da = (right-left)*6*u*(1-u)/2
            c = a+b/16+da/16
            check("paths", f"exact lift {left,right,b,u}", F(11,4) <= a <= 5 and F(21,8) <= c <= F(43,8) and da == -(16*(a-c)+b), "exact finite samples")
metrics["physical_time_example"] = dict(active_T=10, nu=1, tau_max=2, fixed_overhead=4, expected_upper_bound=54)
check("clock", "expected physical time bound example", 10+2*(2*1*10)+4 == 54)

# A weighted finite filter, including its equal-energy sharpness case.
weights = np.array([2., 1., 4., 7.]); energies = np.array([-1., 0., 3., 4.])
weighted = weights*np.exp(-energies)
leak = weighted[2:].sum()/weighted.sum()
upper = 11*math.exp(-3)/(3+11*math.exp(-3))
check("extensions", "weighted filtering inequality", 0 < leak <= upper, "floating point")
sharp = weights*np.exp(-np.array([0., 0., 3., 3.]))
check("extensions", "filter bound sharp at energy limits", close(sharp[2:].sum()/sharp.sum(), upper), "floating point")
check("negative", "omitting multiplicity invalidates bound", upper > math.exp(-3)/(1+math.exp(-3)), "floating point")
check("extensions", "empty rejected set has zero leakage", F(0)/(F(3)+F(0)) == 0)
decoder = {"AAA": "K", "AAG": "K", "GGU": "G"}
section = {p: min(c for c in decoder if decoder[c] == p) for p in ("K", "G")}
check("extensions", "finite coding section", all(decoder[section[p]] == p for p in section))
check("negative", "historical code is not recoverable", decoder["AAA"] == decoder["AAG"] and "AAA" != "AAG")
check("negative", "outside decoder image no section", "W" not in decoder.values())
# Resource-accounted toy transition: each residue consumes one packet.
state = ("", "", 3); total = 3
for codon in ("AAA", "GGU", "AAG"):
    old = state
    state = (old[0]+codon, old[1]+decoder[codon], old[2]-1)
    decoded = "".join(decoder[state[0][i:i+3]] for i in range(0, len(state[0]), 3))
```

```
    check("extensions", "compatible toy extension "+state[1], decoded == state[1] and state[2]+len(state[1]) == total)
check("negative", "mismatched shared interface detected", ("AAA", "K", 2) != ("AAA", "K", 3))
# A matching shared code/product pair supplies neither fresh inputs nor guards.
shared_output = ("AAA", "K")
shared_input = ("AAA", "K")
fresh_packets = 0
check("negative", "shared coordinates do not create fresh resources",
      shared_output == shared_input and not fresh_packets >= 1)
actual_guard_state = {"fresh_packets": 1, "catalyst_present": False}
full_guard = (actual_guard_state["fresh_packets"] >= 1
              and actual_guard_state["catalyst_present"])
check("negative", "shared coordinates do not imply full guard",
      shared_output == shared_input and not full_guard)
# Distance-tube arithmetic does not alone certify container membership.
rho, delta, dx, dh = F(1,4), F(3,4), F(1,8), F(1,8)
check("extensions", "retention perturbation budget", delta-dx-dh > delta/3 and dx < delta/3 and dh < delta/3)
check("negative", "container condition indispensable", 0 < F(1,10) < 1 and not 0 < F(-1,10) < 1)
# A nontrivial quadratic residual witness with explicit exact norm square.
A = np.diag([2., 4.]); yhat = np.array([.1, -.2]); star = np.array([.05, -.05])
grad = A@(yhat-star); residual = np.linalg.norm(grad); m, radius = 2., .5
distance = np.linalg.norm(yhat-star); gap_energy = .5*(yhat-star)@A@(yhat-star)
check("certificates", "quadratic residual hypotheses and conclusions", residual < m*radius and distance <= residual/m and gap_energy <= residual**2/(2*m), "floating point")
check("certificates", "Hessian error budget exact", F(3)-F(1,10)-F(1,2)*F(1,2) == F(53,20) > 0)
check("negative", "positive central Hessian alone insufficient", F(1)-F(1,10)-F(10)*F(1,2) < 0)
# g(t)=1-|t-1/2| has L=1, h=1/4 and sampled minimum=1/2.
check("certificates", "sampled path exact lower certificate", F(1,2)-F(1)*F(1,4)/2 == F(3,8) > 0)
check("negative", "positive endpoint samples miss collision", 1-8*F(0)*(1-F(0)) > 0 and 1-8*F(1)*(1-F(1)) > 0 and 1-8*F(1,2)*(1-F(1,2)) < 0)


def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("--json", default="results_article103.json")
    args = parser.parse_args()
    counts = Counter(c["group"] for c in checks)
    failures = [c for c in checks if not c["passed"]]
    result = dict(article=103, scope="Finite reduced-model verification; not a general proof or chemical validation", versions=dict(python=platform.python_version(), numpy=np.__version__, scipy=scipy.__version__), tolerance=dict(atol=ATOL, rtol=ATOL), deterministic=True, source_sha256=hashlib.sha256(Path(__file__).read_bytes()).hexdigest(), check_count=len(checks), passed=len(checks)-len(failures), failed=len(failures), groups=dict(sorted(counts.items())), metrics=metrics, checks=checks)
    Path(args.json).write_text(json.dumps(result, indent=2)+"\n")
    print(f"Article 103: {result['passed']}/{len(checks)} checks passed")
    print(json.dumps(result["versions"], sort_keys=True))
    for group, count in sorted(counts.items()):
        print(f"  {group}: {count}")
    print("Capture law [empty,target,competitor]:", metrics["capture_law"])
    print("Release law [empty,target,competitor]:", metrics["release_law"])
    print("Source SHA256:", result["source_sha256"])
    for failure in failures:
        print("FAIL:", failure["group"], failure["name"])
    return bool(failures)


if __name__ == "__main__":
    sys.exit(main())
```