Supplementary Material S1

Electronic Companion

Equivalent Optima, Different Policies:
Certified Distribution-Preserving Reformulations
for AI Decision Systems

Supplementary Material S1 specifies the implementation, designed instances, validation boundaries, and reproduction procedure accompanying the main article. Mathematical proofs are retained in the main article. The companion does not add unreported field observations or claim formal verification of the Python implementation.

1 Contract-to-implementation traceability

The release separates three operations. A compiler proposes a bounded representation and reference tables. A verifier re-derives its declared structure and checks consumer-specific obligations. A numerical consumer evaluates a probability law using the checked weights. A valid matrix reformulation alone does not make an arbitrary solver a sampler from that law.

Table 1: Traceability from a scientific statement to executable evidence.

Scientific obligation	Implementation	Evidence
Finite reference transport and weighted spectra	`certificates.py`	Exact rational tests and unequal-spectrum witnesses
Joint-KL separation and correction	`compiled_runtime.py`; learning objectives	Complete-encoding KL checks and matched corrected runs
Compatible layer composition	`compose` in `certificates.py`	Reference/feature mismatch rejection and 200 finite chains
Convex-class reference lift	Linear inequalities and equality checks in `certificates.py`	Feasible vertex lifts; constant-positive-penalty control
Bounded source transformation	`structured_compiler.py`	50 heterogeneous-rule source/emitted-model comparisons
Separate consumer decisions	`verify_product.py`	Distribution-only example exits successfully for physical use and unsuccessfully for joint-KL use

The complete finite checker assumes that its supplied encoded support is complete. The product verifier replaces global enumeration with independent local support reconstruction for exactly the five implemented rules. It does not accept an arbitrary unverified support oracle. It also does not solve global feasibility: its certificate applies to every nonempty decoded-only feasible subset, and a separate witness or solver must establish nonemptiness for a new source instance. Proposition 5 in the main article distinguishes its witnesses: unequal conditional residual masses disprove the distribution contract, whereas two distinct positive-weight atoms in a common spectrum disprove the universal joint-KL contract. The current Python reference-transport validation scans assignments per physical state, costing $O(N|X|)$ in addition to $O(Nk)$ residual arithmetic; output sorting uses $O(N \log N)$ key comparisons.

2 Source fragment and certificate format

The source document has schema identifier `272-bounded-v1`. Required fields are `upper`, `rules`, and `cost`. Optional fields are `auxiliary_bits` and `constraints`. Each physical capacity is an integer from zero to 16; each scored auxiliary count is from zero to four. One representation rule is specified per physical coordinate. Each constraint has exactly three keys: `op`, `a`, and `b`. The operator is `eq` or `le`. Rational coefficients are integers or fraction strings such as `"3/2"`; binary floating-point input is not an exact certificate representation.

The following complete two-coordinate input has physical support $\{(0, 2), (1, 1)\}$:

```
{
  "schema": "272-bounded-v1",
  "upper": [2, 2],
  "rules": ["split", "unary"],
  "cost": ["3/2", "-2/3"],
  "auxiliary_bits": [0, 0],
  "constraints": [
    {"op": "eq", "a": [1, 1], "b": 2},
    {"op": "le", "a": [1, -1], "b": 1}
  ]
}
```

For the split coordinate, (y, z) has decoder $x_1 = y + z$ and guard $y + z \leq 2$. For the unary coordinate, (b_1, b_2) has decoder $x_2 = b_1 + b_2$. Pulling back the two physical constraints gives $y + z + b_1 + b_2 = 2$ and $y + z - b_1 - b_2 \leq 1$. The emitted cost is $(3/2)(y + z) - (2/3)(b_1 + b_2)$. The compiler additionally supplies the normalized local conditional tables. It neither removes a genuine machine identity nor infers physical equivalence from equal cost.

The product certificate contains the entire source, a SHA-256 digest of its canonical JSON representation, transformed integer-variable bounds, decoder and score matrices, pulled-back objective coefficients, transformed constraints, and one complete local conditional table per coordinate. An external caller can bind verification to an expected source, so editing the source and recomputing its digest does not by itself authorize a different problem. The digest is an integrity aid, not a cryptographic signature or proof of authorship.

2.1 Verifier procedure

1. Validate the supported schema, field dimensions, rational numbers, integer bounds, and rule identifiers. Reject unknown source semantics rather than ignoring them.
2. Compare the embedded source to the expected source, when supplied, and verify its canonical digest.
3. Reconstruct the decoder matrix, integer bounds, objective substitution, score matrix, and all transformed guards from the source. Compare them to the emitted model with type-sensitive structural equality: Boolean and floating-point substitutions for exact integer fields are rejected even when Python would otherwise compare their values as equal.
4. For every coordinate and every local physical value, independently enumerate raw local variable bounds, apply the decoder and guards, and recover the complete valid encoding set.
5. Check that the certificate lists exactly that set, with positive rational masses summing to one. Recompute each residual score and aggregate masses with equal residuals.
6. Compare the resulting spectra over local physical values. If they vary, return *inconclusive for the product method*; do not assert global inequivalence.
7. If all local tests pass, issue a distribution-preservation verdict. Issue a separate parameterwise joint-KL verdict according to whether every common local spectrum is a point mass.

The verifier ignores untrusted stored pass flags. Its source-derived matrix routine is shared with the compiler, so systematic errors in that routine remain within the trusted implementation boundary. The independent raw-assignment tests explicitly exercise that boundary; they do not turn the shared routine into a proof-assistant-verified transformation.

3 Runtime and stable objective correction

For a verified product certificate, the reference runtime enumerates physical decisions subject to an explicit size limit. It accepts finite natural parameters and, optionally, finite log reference weights. These weights are normalized on the physical support. For each local common residual spectrum σ_i, it evaluates

$$\begin{aligned}\psi_i(\theta) &= \log \sum_r \sigma_i(r) e^{\theta^\top r},\\ g_i(\theta) &= \theta^\top \mathbb{E}_{\sigma_{i,\theta}} R - \psi_i(\theta),\\ \nabla g_i(\theta) &= \mathrm{Cov}_{\sigma_{i,\theta}}(R)\theta.\end{aligned}$$

Log-sum-exp calculations are used for finite numerical evaluation. The returned quantities include the physical law, expected cost, physical KL, joint KL, auxiliary gap, its gradient, and both learning objectives. Sampling first selects a physical decision from its law and then selects each local encoding with probability proportional to its reference mass times its residual exponential score. This produces the complete encoded law specified in the article; samples are not obtained by uniform calls to an optimization solver.

The correction is evaluated through the simplified identity $J_{\mathrm{corr}} = J_X$, not by subtracting two large floating-point numbers. The encoded objective and its penalty remain available for diagnosis. An optional independent auxiliary parameter block has a different parameterization and must not be confused with correcting the original tied objective.

The reference runtime's physical enumeration cap is two million Cartesian candidates, and complete encoded expansion for tests has a separate one-million-encoding cap. These are explicit safeguards, not mathematical complexity results. The scheduling dynamic program bypasses physical enumeration only for its declared sum/ramp structure. The product theorem allows arbitrary decoded-only coupling constraints, but general inference under those constraints may still be expensive.

4 Designed inputs and units of analysis

4.1 Physical scheduling and distribution comparison

The original scheduling generator imposes $0 \le x_t \le U$, integer production, total output D, adjacent ramp bound R, and an initial ramp from x_0. For the small comparison, $T = 4, 6, 8$, $U = 4$, $D = 2T$, $R = 2$, and $x_0 = 2$. There are five seeded price vectors at each horizon, each evaluated at five inverse temperatures $0.25, 0.5, 1, 2, 4$. This is 15 underlying cost/scheduling instances and 75 matched instance–temperature cases, not 225 independent sampled environments. Costs use a linear price plus $0.08x_t^2$.

The three compared laws are $p \propto e^{-\beta c}$, $q_+ \propto m e^{-\beta c}$, and $q_- \propto m^{-1} e^{-\beta c}$, with $m(x) = \prod_t (x_t + 1)$. The last law represents canonical restriction without reference rebasing. A correctly rebased canonical representation is not expected to fail. The normalized-lift formula equals the physical formula by construction; independent complete-encoding checks establish that the implementation actually realizes the aggregation identity on small instances.

4.2 Two distinct large-support experiments

The large inference study has $T = 8, 16, 32, 64$, $U = 8$, $D = 4T$, $R = 2$, and $x_0 = 4$. Its 64-period physical support contains

$$21{,}089{,}475{,}759{,}356{,}321{,}513{,}097{,}655{,}399{,}891{,}517{,}939{,}291$$

schedules. The large compiler study uses the same bounds, demand, and adjacent ramp limits but *no initial-ramp constraint*. Its 64-period support contains

$$30{,}179{,}055{,}564{,}379{,}251{,}685{,}182{,}525{,}041{,}807{,}117{,}407{,}933$$

schedules. These exact counts are independently computed by integer-state recurrences and retained in CSV. They are not quantities inferred from floating-point partition functions. The constant schedule at four is a feasible witness in both cases.

The compiler study evaluates split, binary, unary, and slack representations at four horizons. Each compilation and verification is timed three times. The timer is `time.perf_counter`; the median is reported. No random-effects uncertainty interval is constructed from three repeats. Count calculation, source JSON formatting, execution of an optimizer, and physical sampling are not included in certificate checking time. Compact certificate byte counts include the emitted dense matrices.

4.3 Contextual learning

Five seeds each generate a three-dimensional Gaussian context design with 192 contexts and a centered 3×6 coefficient matrix. The first 128 contexts train the model; the last 64 are held out. The teacher law is specified exactly, so full-distribution cross entropy can be optimized without drawing labels. The teacher matrix is withheld from the numerical optimizer. Four implementations differ only in the representation-induced schedule offset. This is a controlled representational-capacity test, not a claim about predictive accuracy on real operating records.

4.4 Tied auxiliaries and corrective controls

For seed indices $s = 0, \ldots, 19$, NumPy generator seeds are $272700 + s$. Six prices are Gaussian with standard deviation 0.65 and are centered within the vector. The support is $T = 6, U = 4, D = 12, R = 2, x_0 = 2$. Three values $\tau = 0.5, 1, 2$ and five methods give 300 runs: physical, one tied bit per coordinate, four tied bits, corrected four-bit joint objective, and four-bit untied objective. This is an extension of the original 60 unit-temperature runs, not 360 unique new runs.

All physical parameters start at zero. In the untied case, six extra auxiliary parameters also start at zero, which is their reference-distribution minimizer. PyTorch L-BFGS uses a strong-Wolfe line search, `max_iter=250`, `tolerance_grad=1e-10`, and `tolerance_change=1e-14`. The covariance-based gradient and exact conditional penalty are also tested separately from the optimizer. Stationarity residuals do not prove a global objective gap for a nonconvex parameterized problem.

At unit regularization, approximate nominal 95% percentile bootstrap intervals for mean and median TV resample the 20 cost vectors 10,000 times with NumPy generator seed 272221. The five method groups are processed in the original summary order. Within each method, the same index draws produce its mean and median intervals; quantiles use linear interpolation. This preserves the original mean intervals and adds a median interval from the held CSV. For four tied bits the median is 0.24468194, with interval $[0.22350491, 0.25405542]$. The cost vector is the resampling unit, and the interpretation is conditional on the designed pseudorandom cost generator. The same cost instances appear across methods and temperatures.

4.5 Globally solved replicas

The replica study retains the same constrained support and costs, but uses the nonuniform reference $\mu_x \propto e^{-0.03\sum_t x_t^2}$. Three temperatures and $k = 0, 1, 4$ give 180 analytic solutions. At unit temperature, 60 simplex-logit optimization calls cross-check these laws. They initialize at the analytic natural logits plus seeded Gaussian noise of scale 0.05, use L-BFGS with 200 iterations, gradient tolerance 10^{-11}, change tolerance 10^{-14}, and a strong-Wolfe line search. This near-solution cross-check is not presented as an optimization-algorithm comparison. Global optimality follows from the explicit nonnegative KL difference in the proof.

5 Measures and numerical accuracy

Total variation is $\frac{1}{2}\sum_x |q_x - p_x|$. Unless a table states otherwise, KL is $\mathrm{KL}(q\|p)$ in nats. Contextual learning instead reports $\mathrm{KL}(p_{\mathrm{teacher}}\|q)$. All complete joint TV calculations use enumerated physical supports. The large dynamic program reports KL and moments through additive factor identities; it does not infer joint TV from marginal TV.

For a numerical method, physical cost excess and regularized-objective excess are different quantities. A distribution can have lower expected cost while differing from the intended entropy-regularized policy, so a negative cost difference does not refute a representation error. The raw tables report signs without turning every distortion into a claimed cost deterioration.

The original Python floating-point experiments use float64 and one PyTorch thread. The reference run used Python 3.13.5, NumPy 2.3.5, SciPy 1.17.0, PyTorch 2.10.0+cpu, and Matplotlib 3.10.8 on Linux x86_64. The shared host reported an AMD EPYC 9V74 processor. Timings are local measurements, not dedicated-machine performance claims. Exact certification uses Python's standard-library fractions and integers and is independent of numerical tolerance settings.

6 Validation and deliberate rejection cases

The executable evidence has several independent roles, which are not summed into an inflated count of research observations.

Table 2: Validation groups in the released implementation.

Group	Coverage
Primary finite suite	8,532 assertions spanning reference transport, spectra, conditioning, convex lifts, chain KL, bounds, and analytic scalar optimization
Additional enumeration	18 checks over 819 distinct complete assignments; 200 gradient comparisons; eight malformed-input cases
Layer composition	200 feature-aware chains and a same-mean/same-variance rejection example
Bounded compilation	15 small sources, 120 full-law comparisons, 16 scaling cases, and 11 malformed-table/source mutations
Independent linear emission	50 heterogeneous-rule sources, 2,523 raw assignments, 252 feasible encodings, six additional model/source mutations, and zero-capacity handling
Compiled numerical consumer	80 complete-law/KL comparisons and 800 sampled-decoder checks
Scope and strict input regressions	61 linear-projection checks; 61 nonlinear-curve checks; 183 constant-penalty controls; separate inconclusiveness and nonemptiness cases; 94 rejected type/reference mutations

Rejected or restricted cases include unnormalized or zero reference weights; floating-point weights

presented as exact rationals; Boolean values masquerading as integer encodings; omitted or repeated encodings; incorrect residual vectors; stale source digests; changed decoder or score matrices; modified bounds, objectives or constraints; and mismatched expected source documents. A distribution-only pass must not be escalated to a joint-KL pass. Failure of the product sufficient condition is inconclusive rather than proof that all globally feasible encodings fail. Encoded-only restrictions require renewed checking even when every physical decision retains a representative.

Zero-capacity unary encoding is explicitly tested as a single empty encoding. Point-mass residuals may be nonzero common score shifts. A nonzero conditional penalty may be harmless when constant over a convex physical policy class. Separately parameterized auxiliaries may optimize their gap without changing the physical policy. These controls protect against stronger conclusions than the theorems support.

6.1 Parameter-domain and exact-input regressions

The additional standard-library script `Code/scope_regression_checks.py` checks the distinctions stated inside the main theorems. A two-dimensional layer whose residual spectra are $\delta_{(0,0)}$ and $\delta_{(0,1)}$ fails the ambient test but passes after the linear restriction $\theta_2 = 0$. This is the projected-feature contract, not a false positive from a numerical parameter test.

For a separate nonlinear example, let

$$\sigma_0 = \tfrac{1}{2}\delta_{(-1,0)} + \tfrac{1}{2}\delta_{(1,0)}, \qquad \sigma_1 = \delta_{(0,1)}.$$

Their exponential transforms are $\cosh\theta_1$ and e^{θ_2}. They agree along $\theta_2 = \log\cosh\theta_1$ without being the same finite measure. The exact checker rejects the ambient contract; 61 points check the curve calculation, and an off-curve point shows a physical-probability discrepancy. The algebra establishes the counterexample, not the number of sampled points.

Constant-positive conditional KL is checked in 183 analytic binary controls. A local product test is also made inconclusive while the explicit globally restricted support passes; a different infeasible source confirms that product certification does not solve nonemptiness. Exact-input regressions reject 80 value-preserving Boolean/float substitutions in emitted integer model fields, ten such expected-source substitutions, and four malformed reference/decoder inputs. These checks close a software-type validation gap without changing any mathematical model or enlarging the research sample.

6.2 Equation-derived policy-surface figure

`Code/make_policy_surfaces.py` evaluates Equation (14) on 19,642 parameter pairs, with $a \in [-4, 4]$, $\tau \in [0.35, 3]$, and $k = 3$. The plotted grid and optional Wolfram source are in `Figure_Data`. `Code/test_policy_surfaces.py` runs independent decimal, scalar-optimization, and complete-assignment checks, recorded separately from the original assertion counts. The supplied vector figures are calculated in Python; no generated artwork or Mathematica execution is used.

7 Reproduction commands and archived outputs

The commands below require the separate code-and-results archive retained with this study. The replication material is available from Zenodo at `https://doi.org/10.5281/zenodo.23253837`. That archive is not attached to this companion. S1 documents its contents and use. Its package root is independent of an absolute user-drive path. The exact checker requires only Python; numerical work additionally uses the pinned libraries in `requirements.txt`. Run:

```
python Code/reproduce_revision.py --verify-saved
```

The `--verify-saved` flag checks saved digests before regenerating numerical outputs; the command runs the supported validation stages and compares deterministic statistical columns against the held results. It writes logs and a machine-readable report. The Zenodo replication package excludes manuscript sources and does not support the `--with-pdfs` option. Article and companion PDFs can be compiled from the separate editable-source bundle using a LaTeX installation containing the declared packages and the supplied vector figures; figure regeneration and Ghostscript are not required. Figure-generation scripts remain available separately. Wall-clock measurements and PDF binary digests vary across environments.

The reproduction comparison keeps exact integers, schemas, references, and rational certificates exact. Iterative tied-parameter optimizer endpoints use the narrowly specified comparison allowance documented in the software; that allowance is not a theorem tolerance or a global-optimality guarantee. Separate regression tests protect unchanged controls, finite outputs, and the printed Table 8 statistics.

A bounded source can be compiled and verified independently:

```
python Code/structured_compiler.py \
  Code/Specs/production_tied.json \
  --output Results/recompiled_tied.json
python Code/verify_product.py \
  Code/Specs/production_tied.json \
  Results/recompiled_tied.json --consumer physical
python Code/verify_product.py \
  Code/Specs/production_tied.json \
  Results/recompiled_tied.json --consumer joint-kl
```

The `--consumer joint-kl` command intentionally exits with status three: physical inference passes but the uncorrected joint-KL contract does not. This is an expected consumer-specific failure, not a failed installation. Exit two indicates invalid input. Once the archive is obtained, running the examples and scripts requires no external API, private data service, or paid solver.

8 Additional reference experiments

The added studies are standalone JavaScript implementations. The archived optimizer calculations used IEEE-754 double precision in an execution environment whose engine build and processor were not recorded. Their source and endpoints are retained in the separate archive for checking. Revised direct-check timings were measured in Node.js 24.16.0 (V8 13.6.233.17-node.49), Windows x64 build 10.0.26200, on an Intel Xeon Gold 6430 host reporting 64 logical processors. They use the monotonic `performance.now()` clock. Original millisecond-clock timings remain archived separately; the new timing file records the measured environment.

8.1 Optimizer and initialization design

The scheduling law and purely linear cost are the same mathematical model as the tied-auxiliary experiment, evaluated through separate probability sums and an analytic covariance gradient. For $p_\theta(x) \propto e^{\theta^\top x}$ and $h_\theta(x) = c(x) + \tau\theta^\top x$, the physical gradient is $\mathrm{Cov}_{p_\theta}(X, h_\theta)$. The k tied Bernoulli auxiliaries together contribute $\tau k\theta_t r_t(1 - r_t)$ to coordinate t, with $r_t = \mathrm{sigmoid}(\theta_t)$. Central differences check the implementation. For $k = 0$, $\theta = -a/\tau$ supplies an analytic physical optimum.

The main factorial design uses five cost vectors, $\tau = 0.25, 0.5, 1, 2, 4$, $k = 0, 1, 4$, three starts and two algorithms, for 450 executions on the 387-schedule six-period support. Mulberry32 uniforms at midpoints of their finite bins feed a Box–Muller transform. Gaussian prices have standard deviation 0.65 and are centered. Cost seeds are $272900 + 100T + s$; the two nonzero starts have standard deviation 0.35 and seeds $273100 + 1000T + 100s + i$, $i = 1, 2$. These seeded designs do not justify independent-sample or field-population inference.

Both algorithms use Armijo backtracking with coefficient 10^{-4}, initial step one and 45 halvings at most. BFGS uses a dense inverse-Hessian update; Polak–Ribière+ conjugate gradient restarts its direction every T iterations and resets to steepest descent when its proposed direction is not sufficiently descending. The gradient infinity-norm target is 10^{-7} and the initial iteration budget is 1,200. All fifteen primary misses are restarted from their endpoints with fresh optimizer state and a further 10,000-iteration budget. Initial statuses and final continuation records are both retained. Three primary conjugate-gradient endpoints remain above threshold; their largest final residual is 2.0063×10^{-7}. No global convergence claim follows.

The size/auxiliary study uses $T = 4, 6, 8$, three costs at each horizon, $\tau = 1$, $k = 0, 1, 2, 4, 8$, zero starts and BFGS. Nine of its 45 cells reuse main-design endpoints, leaving 36 additional initial executions. One eight-period/eight-bit cell stops at 1,200 iterations with residual 1.2889×10^{-7}. All results are retained. The three support sizes are 31, 387 and 5,231. Costs vary with T; the study does not isolate a causal size effect.

Only the version-3 optimizer source with the mixed generator is used for the reported estimates. Earlier pilots used adjacent xorshift32 seeds with correlated early outputs and are excluded. The version-3 JSON retains every cost vector, start, endpoint, objective, gradient residual, TV, evaluation count and stopping status.

8.2 Exact spectrum and direct-check comparison

The explicit-support study uses positive integer reference weights with a common denominator, integer scalar features and deterministic decoding. Its exact checker compares masses by BigInt cross-products. It is a specialization for the supplied finite inputs, not a general rational parser. The numerical path independently normalizes physical and complete encoded exponential weights and aggregates through the decoder. Input construction is excluded from per-call timing; the measured algorithms include their own allocations and arithmetic.

The designed finite-grid counterexample uses the conditional spectra

$$\sigma_0 = (6\delta_0 + 57\delta_2 + 32\delta_4 + \delta_6)/96, \qquad \sigma_1 = (29\delta_1 + 58\delta_3 + 9\delta_5)/96.$$

Their transform difference is $(t-1)^4(t-2)(t-3)/96$, $t = e^\theta$. The selected grid $\{0, \log 2, \log 3\}$ therefore agrees exactly despite unequal spectra. The order-four root at $t = 1$ gives matching raw moments through order three. Other controls include a nonzero point residual, a common multi-atom spectrum, and unequal spectra with matching low-order moments. These deliberately selected controls establish possible failures of simpler diagnostics, not their prevalence.

Twelve timing cells cross $|X| = 16, 64, 256, 1024$ with $M = 2, 8, 32$ residual atoms; $|Z| = |X|M$. Each direct check evaluates one, nine or 33 parameters. Three warm-up calls precede doubling repetitions until a calibration batch reaches 25 ms or 65,536 repetitions. Seven measured batches follow, and the median duration divided by repetitions is reported to two significant figures. Fixed execution order, allocation, JIT state and host noise limit interpretation of small differences.

A separate nine-cell design evaluates objective and gradient correction at five parameters per cell. The common-spectrum penalty uses $g = \theta\psi' - \psi$ and $g' = \theta\psi''$. Explicit subtraction checks $J_Z - \tau g = J_X$ and the corresponding scalar-gradient identity. Symbolic reduction simply reuses the physical evaluator; neither identity nor the isolated component timings establish an end-to-end training speedup.

8.3 Additional reproduction commands

From the separate archive root, the standalone reference files can be run with a modern Node.js installation:

```
node Code/p272_optimizer_reference_v3.js \
  > Results/p272_optimizer_reference_results_v3.json
node Code/p272_directcheck_reference.js correctness \
```

```
  > Results/p272_directcheck_correctness.json
node Code/p272_directcheck_timing_node.js timing \
  > Results/p272_directcheck_timing_reproduced.json
```

The optimizer script writes raw endpoints and summaries in one run. The new timing script differs from the archived reference source only in its clock, runtime metadata and provenance comments; its check algorithms are unchanged. Timing JSON retains all seven batches per metric. These Node commands are separate from the Python validation runner. Keep the held outputs when comparing a reproduction run, and retain numerical threshold misses as reported outcomes.

9 What the release does not certify

The package does not establish a universal first-in-literature claim, a proof-assistant-checked implementation, superiority over WAPS/ProCount/Dice/FLARE executables, or readiness for a safety-critical deployment. No such executables were benchmarked here. The formal guarantees are the declared finite theorems; the software evidence comprises reproducible internal tests and independent calculation paths. The source-bound checker cannot repair an incorrect real-world decoder or infer the intended reference policy. These scope limits are part of the scientific result, not evidence of field deployment.