Shunyaya Symbolic Mathematical Symbols (SSMS) – v1.8

Core Operator Layer over SSM

Status: Public Release (v1.8)
Date: 01 October 2025

ABSTRACT

Symbolic Symbols: where the operators themselves become symbolic, carrying stability without changing classical results.

Imagine arithmetic that preserves every classical result yet carries a bounded "stability" signal through every operation, so calculations not only compute but also disclose how calm or stressed they are.

Shunyaya Symbolic Mathematical Symbols (SSMS) is the **operator canon**: a compact, portable set of **alignment-aware operators** that act on symbolic numerals to produce results with both classical magnitude and a bounded alignment. SSMS preserves **collapse parity** with classical math, enforces **bounds** on alignment, supports **streaming-safe addition**, offers **banded comparisons** for decision confidence, and provides an optional **environment gate** that attenuates alignment only (**a_env = g_t * a_op**), leaving magnitudes untouched. In short, SSMS supplies the **verbs and grammar** that make stability-aware arithmetic usable in real workflows.

Shunyaya Symbolic Mathematics (SSM) provides the numeral and the collapse map:
x = (m, a) # magnitude m, alignment a in (-1, +1)
phi(m, a) = m # collapse recovers classical results

SSMS complements SSM by defining portable operators (s_add, s_mul, s_div, s_pow, s_gt, s_eq, min/max/abs/round). Two combine policies keep alignment bounded and composable:

- **M1 (simple combine):** a' = a1 * a2
- **M2 (rapidity combine):** a' = tanh(atanh(a1) + atanh(a2))

This yields a brief, implementation-ready canon where:

1. **Collapse safety** holds by construction (every SSMS result reduces to its classical value under phi).
2. **Alignment** travels alongside magnitude to expose stability, sensitivity, and decision confidence without changing classical outcomes.

3. **Portability** is immediate: vectors, matrices, and polynomials compose from the same operators; optional gating calms only the alignment channel.

What this document delivers:

An **ASCII-only operator spec**, a **minimal manifest** for reproducibility, and a few **micro examples** showing how **stability-aware arithmetic** clarifies **averaging**, **matrix multiply**, **scoring**, and **banded decisions** — making **SSMS** the natural, complementary bridge from **SSM** to real-world, composable workflows.

Caution: Observation-only. SSMS provides stability-aware arithmetic for analysis and decision support; it does not replace domain models, operational controls, or safety processes.

For the numeral (m, a), the collapse map phi, and canon details, see **Shunyaya Symbolic Mathematics — Detailed (v2.3)** or the **SSM Brief (Numerals & Formation)** for a quick recap.

Benefits & Complementarity (at a glance)

• **Classical compatibility:** collapse parity holds for every operator (**phi** preserves all classical results; includes zero/identity determinism: **phi(0, a) = 0** and **phi(1, a) = 1**).

• **Stability signal:** a bounded alignment **a in (-1, +1)** travels with magnitude, exposing calm vs stress without changing classical outcomes.

• **Calm decisions:** numeric **s_gt** and **s_eq** scores support banded **true / false / undecided** reporting for confidence-aware choices.

• **Streaming-safe sums: s_sum** uses explicit accumulators **(U, W)** so addition remains associative in streaming and batch.

• **Safe composition:** the same operators build scalars, vectors, matrices, polynomials, plus practical helpers (**min, max, abs, round**).

• **Predictable combine laws (bounded by design):** alignment stays bounded under chaining using portable policies — **M1: a' = a1 * a2**; **M2: a' = tanh(atanh(a1) + atanh(a2))**; with clamp guard **|a| <= 1 - eps_a**.

• **Environment awareness (optional):** apply a gate to alignment only (**a_env = g_t * a_op**); magnitudes stay unchanged.

• **Reproducibility:** a minimal **manifest** declares **clamps, weights, and gate knobs** for audit-ready runs.

• **Explainability hooks: (U, W)**, **g_t**, and **s_gt/s_eq** yield an audit trail for why a result is calm vs stressed.

• **Natural complement:** Shunyaya Symbolic Mathematics (**SSM**) provides the numeral **(m, a)** and collapse **phi**; Shunyaya Symbolic Mathematical Symbols (**SSMS**) supplies the portable operator canon that makes stability-aware arithmetic usable in real workflows.

Continuity note (for readers new to SSM).

SSM defines the numeral `x = (m, a)` with **collapse** `phi(x) = m`; **SSMS** supplies the verbs that act on these numerals. Intuitively, treat `a in (-1, +1)` as a stability dial. We work in rapidity space via `u = atanh(clamp_a(a, eps_a))`, combine there, then decode with `a = tanh(u)` so alignment stays bounded while magnitudes behave exactly like classical math.

- **Sums (example).**
  `m_sum = m1 + m2`
  `u_sum = (w1*u1 + w2*u2) / max(w1 + w2, eps_w)`, with `w_i = |m_i|^gamma` and `u_i = atanh(clamp_a(a_i, eps_a))`
  `a_sum = tanh(u_sum)`
- **Products (example).**
  `m_mul = m1 * m2`
  Bounded combine policies:
  `M1: a' = a1 * a2`
  `M2 (default): u' = u1 + u2 ; a' = tanh(u')`
- **Powers (integer k).**
  `m_pow = m^k` ; `a_pow = tanh(k * atanh(a))`
- **Comparisons (banded).**
  `dv = (m_x - m_y) / max(|m_x| + |m_y|, V_eps)`
  `du = g_t * (u(a_x) - u(a_y))` *(optional gate `g_t in [0,1]` attenuates alignment only)*
  `s_gt(x,y) = tanh(beta_v*dv + lambda_u*du)`
  `raw_eq = beta_v*|dv| + beta_u*|du|`
  `s_eq(x,y) = 1 - 2*tanh(raw_eq)`
  Banded report:
  ```
  if s_eq >= tau_eq -> "equal"
  else if s_gt >= +tau_hi -> "x > y"
  else if s_gt <= -tau_hi -> "x < y"
  else -> "undecided"
  ```
- **Lifting recipe.**
  To lift any classical `f`, replace `+ -> s_add`, `- -> s_sub`, `* -> s_mul`, `/ -> s_div`, `^ -> s_pow`, and use alignment-aware `min/max/abs/round`. The collapse still returns `f(m, ...)`; the extra `a` channel quantifies calm vs stress.

Soft reminder (observation-only): This note and SSMS are for **analysis and decision support**; they do **not** replace domain models, operational controls, or safety processes. `phi` preserves classical outcomes, and `g_t` never changes magnitudes.

TABLE OF CONTENTS

ABSTRACT 1
SECTION 1. SCOPE & CANON 6
1.1 What SSMS is 6
1.2 Canonical object and collapse 6
1.3 Identities, Inverses, and Zero-Class Display 7
1.4 Pooling Weights (for rapidity means) 8
1.5 Collapse-Safety Contract 9
1.6 Optional Environment Gate (Operator Finalizer) 11
1.7 Minimal Manifest (declare once per study) 13
1.8 Scope, Non-Goals, and ASCII Policy 15
SECTION 2. OPERATORS 15
2.1 s_sum (n-ary, streaming-safe) 16
2.2 s_add / s_sub (pairwise, streaming-safe) 16
2.3 s_mul / s_div (M2 rapidity combine) 17
2.4 s_pow (scalar power r) 18
2.5 s_unary (general monotone transform f) 19
2.6 Unified Alignment Governor (g_t): Placement & Policy 19
2.7 Practical Helpers (min, max, abs, round) 21
2.8 Decision Layer: Banded Comparisons (s_gt, s_eq) 22
2.9 Tensor & Einstein Ops (s_einsum) 23
2.10 Autodiff & Compositionality (u-space recipe) 24
2.11 Performance Modes (big-data safe, exact-by-default) 26
2.12 Linear Algebra Stability Notes (A_beta bound) 27
2.13 Division Policy & Edge Cases (meadow_div) 29
2.14 Polynomials & Horner Bridge (s_horner) 31
2.15 Error Budget Channel (s_error) 32
2.16 Identities, Neutral & Absorbing Elements (quick canon) 33
2.17 Physics Micro-Examples 35
2.18 Finance Micro-Examples 36
2.19 Telecom & Signals Micro-Examples 36
2.20 Universality Note — SSM + SSMS Complement Core Mathematics (applies to every domain) 38

SECTION 3. COMPARISONS & BANDS 39
3.1 s_gt (greater-than score) 39
3.2 s_eq (equality / near-equality score) 40
3.3 Banded Reporting Policy (global) 40
3.4 Band Calibration (global, minimal and reproducible) 42
SECTION 4. BRIDGES 44
4.1 Structural Composition (vectors, dot, matrix multiply) 44
4.2 Polynomials & Functions (composition via s_pow and s_unary) 45
4.3 Decision Flows (scores -> bands -> gate) 46
4.4 Reproducibility & Manifest (cross-layer) 47
SECTION 5. EXAMPLES 49
5.1 E1 - Alignment-Aware Averaging (s_sum) 49
5.3 E3 - s_unary(sqrt) vs s_pow(r = 0.5) 50
SECTION 6. APPENDIX — MANIFEST & QA 51
6.1 Manifest Crib (publish once per study) 51
6.2 QA Checklist (run once per study/release) 53

SECTION 7. FUTURE WORK 55
7.1 Five-Element Transition Layer (deferred) 56
7.2 Domain Adapters (family roadmap) 56
7.3 Mini-Spec Template (2 pages per domain) 56
7.4 Band Calibration Guide (short protocol) 56
7.6 Governance & Licenses 56
7.7 Versioning & Change Policy 57
8. VERSION HISTORY 57
Appendix A — Unified Manifest Skeleton (single source of truth) 57
Appendix B - QA & Release Checklist (Go/No-Go) 59
Appendix C — Glossary & Quickref (minimal, copy-ready) 61
Appendix D — Interactive Demo (single-file, non-normative) 64

SECTION 1. SCOPE & CANON

What this section covers. A crisp foundation for SSMS: the symbolic numeral and collapse, default clamps and guards, identities and inverses, pooling weights, the collapse-safety contract, and an optional environment gate hook. The goal is a minimal, portable canon that preserves classical results under **phi** while enabling stability-aware operations through a bounded alignment channel.

1.1 What SSMS is

Shunyaya Symbolic Mathematical Symbols (SSMS) is the **operator canon** that makes stability-aware arithmetic usable in practice. It defines portable, alignment-aware versions of everyday operations (s_add, s_sub, s_mul, s_div, s_pow, s_gt, s_eq, plus min/max/abs/round) that act on symbolic numerals while preserving classical results under collapse.

Guarantees and intent (at a glance):

- **Collapse parity:** Every SSMS result reduces exactly to the classical value under the collapse map.
- **Bounded alignment:** The alignment channel remains strictly within (−1, +1) and never alters classical magnitudes.
- **Streaming-safe addition:** Summation is defined with explicit accumulators so associativity holds in streaming and batch.
- **Decision readiness:** Comparisons yield numeric scores that support banded "true/false/undecided" reporting.
- **Environment hook (optional):** A calm gate may attenuate only the alignment channel at the operator boundary.
- **Portability:** The same operators compose cleanly for scalars, vectors, matrices, and polynomials.

Scope: SSMS provides the verbs and grammar (operators and guards). The numeral itself and the collapse map are defined next, then referenced by all operators.

1.2 Canonical object and collapse

A symbolic numeral carries a classical magnitude and a bounded alignment:

x = (m, a)
m in R # classical magnitude
a in (-1, +1) # bounded alignment

Collapse (observation-only):
phi(x) = m

Rapidity (linearize alignment safely):
u = atanh(clamp_a(a, eps_a))
a = tanh(u)

Clamp (keep alignment strictly inside bounds):
clamp_a(a, eps_a) = sign(a) * min(abs(a), 1 - eps_a)

Invariants:

- |a| < 1 at all times (by clamp and tanh).
- u is real-valued and unconstrained.
- Collapse never changes classical results (phi forgets alignment).

1.3 Identities, Inverses, and Zero-Class Display

Additive identity.

```
id_add = (0, +1)
```

Multiplicative identity.

```
id_mul = (1, +1)
```

Negation (additive inverse).
Flips magnitude; preserves alignment.

```
s_neg( (m, a) ) = ( -m, a )
```

Multiplicative inverse.
Use rapidity for alignment; magnitude follows the active division policy (strict/meadow/soft).

```
clamp_a(a, eps_a) = sign(a) * min( abs(a), 1 - eps_a )
u(a)               = atanh( clamp_a(a, eps_a) )

s_inv_mul( (m, a) ) = ( inv_m(m) , tanh( -u(a) ) )
```

Magnitude path `inv_m(m)` by policy:

```
# strict (default): domain m != 0
inv_m(m) = 1/m

# meadow (totalized):
inv_m(m) = 0           if m == 0
         = 1/m         otherwise

# soft (guarded):
inv_m(m) = 1 / ( sign_star(m) * max( |m| , denom_soft_min ) )
sign_star(0) = +1
```

Identity laws (collapse-safe).

```
s_add( x , id_add ) = x
s_mul( x , id_mul ) = x
s_div( x , id_mul ) = x
s_add( x , s_neg(x) )  ->  magnitude 0   # display zero-class (0, +1)
```

Right-inverse on magnitudes:

```
# holds in strict or soft, and in meadow when m_x != 0
s_mul( x , s_inv_mul(x) )  ->  id_mul on magnitudes
```

Meadow zero case:

```
# if meadow_div and m_x == 0:
s_mul( x , s_inv_mul(x) )  ->  magnitude 0   # recommend zero-class display
(0, +1)
```

Zero-class display (canonicalization).

```
if m_out == 0: display (0, +1)
# display-only: does not alter internal alignment computations
```

Notes and guards.

- **Bounds.**

```
|a| <= 1 - eps_a      # enforced via clamp + tanh
```

- **Collapse parity.**

```
phi( id_add ) = 0
phi( id_mul ) = 1
phi respects all identity laws by construction
```

- **Division guard.**
  If `division_policy = "strict"`, ensure `m != 0` before calling `s_inv_mul` or placing a zero in the denominator.
  If `division_policy = "meadow"` or `"soft"`, magnitude is totalized/guarded; alignment still uses `tanh(-u(a))`.
- **Gate interaction (optional).**
  Environment gate attenuates alignment only (post-op):

```
a_env = clamp_a( g_t * a_op , eps_a )
phi( (m_op, a_env) ) = m_op
# for m_out == 0 the final display remains (0, +1)
```

1.4 Pooling Weights (for rapidity means)

Definition (choose once per study):
w_i = |m_i|^gamma , with **gamma >= 0** and **w_i >= 0**.

Weighted rapidity mean (used by s_sum and other pools):
U = sum_i [w_i * atanh(clamp_a(a_i, eps_a))]
W = sum_i w_i
mean_u = U / max(W, eps_w)
a_pool = tanh(mean_u)

Collapse and bounds:

- Collapse is unaffected (pooling touches alignment only).
- a_pool always satisfies **|a_pool| < 1** (clamp + tanh).

Domain guards:

- Use **max(W, eps_w)** to avoid division by zero.
- All **w_i >= 0** by construction (no sign flips via weights).

Notes:

- **gamma = 0** gives **equal weights**; **gamma > 0** emphasizes larger |m|.
- Publish **gamma** (and **eps_w**) in the manifest.
- Keep **U** and **W** as streaming accumulators to preserve associativity across batches/windows.

1.5 Collapse-Safety Contract

Definition (for every SSMS operator `s_op`).
Let `phi((m, a)) = m` be the collapse map from a symbolic numeral `(m, a)` to its classical magnitude. For any admissible inputs `x, y, ...`:

```
phi( s_op(x, y, ...) ) = classical_op( phi(x), phi(y), ... )
```

This holds for scalars, vectors, matrices, and tensors (elementwise for maps; structurally for reductions) wherever the corresponding classical operation is defined.

Examples (pointwise).

```
phi( s_add(x, y) ) = phi(x) + phi(y)
phi( s_mul(x, y) ) = phi(x) * phi(y)
phi( s_pow(x, r) ) = ( phi(x) )^r
phi( s_min(x, y) ) = min( phi(x), phi(y) )
```

Why this holds (construction sketch).

- **Magnitudes** follow the classical operation exactly by SSMS design.
- **Alignment** is combined only in linearized stability (rapidity) space and/or via bounded gates; it does **not** feed back into magnitudes.
- Therefore `phi`, which returns magnitudes, reproduces the classical result.

Composability (closure under composition).
If `s_op1` and `s_op2` each satisfy the contract, then for all admissible `x, y, z`:

```
phi( s_op2( s_op1(x, y), z ) )
= classical_op2( classical_op1( phi(x), phi(y) ), phi(z) )
```

By structural induction, the same holds for any finite expression tree built from SSMS operators.

Map/Reduce corollaries.

- **Map:** for any unary `s_f` satisfying the contract,

```
phi( map(s_f, X) ) = map( f_classical, phi(X) )
```

- **Reduce:** for any associative `s_op` satisfying the contract,

```
phi( reduce(s_op, X) ) = reduce( op_classical, phi(X) )
```

Batch/streaming parity follows immediately for reducers like `s_sum` because the magnitude path equals the classical accumulator.

Zero-class neutrality (display vs. collapse).

```
if m == 0: display (0, +1)    # canonical zero-class
phi((0, +1)) = 0              # collapse unchanged
```

Guards and domains.

- **Definedness.** The contract applies **only** where the classical operation is defined. Examples:
  - Division: no division by zero unless `meadow_div == true` (opt-in totalized division).
  - Power: `(phi(x))^r` must be classically defined (e.g., avoid non-integer `r` on negative bases unless your classical library permits it).
- **Clamps.** Alignment clamps and denominator guards are mandatory and **alignment-only**:

```
eps_a = 1e-6      # enforce |a| <= 1 - eps_a
eps_w = 1e-12     # guard for weighted means / denominators
```

These guards never modify magnitudes; they only bound alignment transforms and stabilize pooling.

Environment gate neutrality (optional, alignment-only).
If a gate is used, it attenuates alignment **after** the operator completes and does not touch magnitudes:

```
a_env = clamp_a( g_t * a_op , eps_a )
phi( (m_op, a_env) ) = m_op
```

This preserves the contract with or without gating.

Minimal QA invariants (L0 checks for this section).

```
# L0.1 Collapse parity
assert phi( s_op(x, y, ...) ) == classical_op( phi(x), phi(y), ... )

# L0.2 Alignment bounds (post-op, post-gate if used)
assert |a_out| <= 1 - eps_a

# L0.3 Batch/stream parity (for reducers)
assert phi( s_sum(batch) ) == sum( phi(batch) )
assert phi( fold_stream(s_sum, stream) ) == sum( phi(stream) )
```

1.6 Optional Environment Gate (Operator Finalizer)

Definition (alignment-only attenuator).
Given an operator output `(m_op, a_op)`, optionally apply a gate `g_t in [0,1]` **to alignment only**:

```
a_env = clamp_a( g_t * a_op , eps_a )
m_env = m_op
```

`g_t` may be a scalar or a field broadcastable to the shape of `a_op`.

Example lane recipe (bounded, with memory).

```
clip(x, lo, hi) = min( max(x, lo), hi )

# inputs (each in [0,1])
Z_t  in [0,1]          # lane stress
A_t  = 1 / ( 1 + Z_t ) # instantaneous calm (in [0,1])

# memory accumulator (in [0,1])
Q_t  = rho * Q_prev + (1 - rho) * clip( A_t - Z_t , 0 , 1 )

# gate (pre-clamp)
g_t  = ( 1 / ( 1 + Z_t + kappa * abs( Z_t - A_t ) ) ) * ( 1 - exp( -mu *
Q_t ) )

# clamp to [0,1]
g_t  = clip( g_t , 0 , 1 )
```

Parameters: `rho in [0,1]`, `kappa >= 0`, `mu >= 0`, with published initial `Q_0 in [0,1]`.

Properties.

- **Collapse parity.**

```
phi( (m_op, a_env) ) = m_op
```

- **Bounds.** If `|a_op| <= 1 - eps_a` and `g_t in [0,1]` then

```
|a_env| <= 1 - eps_a
```

- **Monotone calming.**

```
g_t = 1  ->  a_env = a_op
g_t = 0  ->  a_env = 0
|a_env| is nondecreasing in g_t for fixed a_op
```

- **Stateless vs stateful.**
  Set `mu = 0` (or `rho = 0`) for a memoryless gate. Positive `mu` and `rho` produce a gradual, history-aware calming via `Q_t`.
- **Composability (pointwise).**
  Gating happens **after** each operator; repeated gates are safe. With clamping, the effective gain is bounded by the product of the gains.

Guards and manifest keys.

```
gate_used   in {"on","off"}   # if "off": set g_t = 1 everywhere
lane_recipe : <string>        # e.g., "ZE-lane-v1"
kappa, mu   : floats >= 0
rho         : float in [0,1]
Q0          : float in [0,1]  # initial memory
Z_t_source  : <declared>      # how Z_t is computed
```

Operational rules:

- Apply the gate **after** each operator's alignment is computed; **never** change magnitudes.
- Publish that `Z_t, A_t, Q_t, g_t` are clamped to `[0,1]`.
- Ensure `clamp_a` is applied at the end:

```
a_env = clamp_a( g_t * a_op , eps_a )
```

- For tensors, `g_t` must be broadcast-compatible with `a_op`.

Tiny examples (illustrative).

```
eps_a = 1e-6

a_op = +0.80, g_t = 0.50  ->  a_env = clamp_a( 0.40 , eps_a ) = +0.40
a_op = -0.60, g_t = 0.25  ->  a_env = clamp_a( -0.15, eps_a ) = -0.15
```

Minimal QA invariants (L0–L1 for gating).

```
# L0 collapse parity
assert phi( (m_op, clamp_a(g_t*a_op, eps_a)) ) == m_op

# L0 bounds
assert |clamp_a(g_t*a_op, eps_a)| <= 1 - eps_a
```

```
# L1 gate off equals identity on alignment
if gate_used == "off":  assert clamp_a(1*a_op, eps_a) == a_op  (within
numeric tolerance)

# L1 monotonic calming (sampled)
if g1 <= g2:  assert |clamp_a(g1*a_op, eps_a)| <= |clamp_a(g2*a_op, eps_a)|
```

1.7 Minimal Manifest (declare once per study)

Required keys.

```
gamma                 # >= 0 ; weighting exponent for pooling (e.g., w_i =
|m_i|^gamma)
eps_a                 # alignment clamp margin (recommend 1e-6)
eps_w                 # denominator guard for pooling (recommend 1e-12)

division_policy       # "strict" | "meadow" | "soft"   (default "strict")
denom_soft_min        # > 0 ; required iff division_policy == "soft"
meadow_div            # boolean alias for "meadow" (compatibility only;
prefer division_policy)

gate_used             # "on" | "off" ; alignment-only gate applied after
each operator
lane_recipe           # short string id for gate (e.g., "ZE-lane-v1");
required iff gate_used == "on"
kappa, mu             # >= 0 ; gate knobs
rho                   # in [0,1] ; gate memory knob
Q0                    # in [0,1] ; initial memory state for gate
```

Display and streaming.

```
zero_class_display   # boolean (default true; show m == 0 as (0, +1))
init_U                # default 0.0 ; streaming accumulator init for sums
init_W                # default 0.0 ; streaming accumulator init for sums
```

Comparisons and bands (global policy).

```
V_eps                 # > 0 ; magnitude normalizer floor used in s_gt
beta_v                # >= 0 ; weight for normalized magnitude delta in s_gt
lambda_u              # >= 0 ; weight for alignment delta (rapidity) in s_gt
beta_u                # >= 0 ; weight for alignment gap in s_eq

# band thresholds (use quantile calibration once; then freeze)
tau_hi                # in (0,1) ; decision band for s_gt (">" / "<")
tau_eq                # in (0,1) ; equality threshold for s_eq ("equal"
takes precedence)

# calibration bookkeeping (publish for reproducibility)
alpha_gt              # in (0,1)
alpha_eq              # in (0,1)
ref_size              # integer ; |D_ref| used for calibration
split_seed            # integer ; seed for train/holdout split
policy_gate           # "on" | "off" ; gate state used during calibration
(must match gate_used)
```

Compatibility (accepted but not preferred).

```
meadow_div           # boolean ; synonym for division_policy == "meadow"
gate_used_bool       # boolean ; if provided, map true->"on", false->"off"
(avoid in new manifests)
V_unit               # legacy scaler for magnitude scores ; prefer V_eps in
s_gt
tau_eq_mag           # legacy magnitude-only tie band ; prefer s_eq +
tau_eq
tau_align            # legacy alignment floor ; prefer s_gt weights
(beta_v, lambda_u) + bands
```

Example (minimal JSON, strict division, gate off, pre-calibrated bands).


```
{
  "gamma": 0.0,
  "eps_a": 1e-6,
  "eps_w": 1e-12,

  "division_policy": "strict",
  "denom_soft_min": null,
  "meadow_div": false,

  "gate_used": "off",
  "lane_recipe": null,
  "kappa": 0.0,
  "mu": 0.0,
  "rho": 0.0,
  "Q0": 0.0,

  "zero_class_display": true,
  "init_U": 0.0,
  "init_W": 0.0,

  "V_eps": 1e-12,
  "beta_v": 1.0,
  "lambda_u": 1.0,
  "beta_u": 1.0,

  "tau_hi": 0.70,
  "tau_eq": 0.80,
  "alpha_gt": 0.05,
  "alpha_eq": 0.10,
  "ref_size": 1000,
  "split_seed": 1729,
  "policy_gate": "off"
}
```


Notes and guards.

```
- division_policy "strict": require nonzero denominators for division and
multiplicative inverse
- division_policy "meadow": totalized inverse on magnitudes (inv_m(0) = 0)
- division_policy "soft": guard tiny denominators with denom_soft_min > 0 ;
preserve sign

- gate_used must be consistent with policy_gate used during band
calibration
```

```
- gating never alters magnitudes; apply clamp_a at the end of the alignment
path
- publish all numeric knobs actually used in a release; freeze for
reproducibility
```

1.8 Scope, Non-Goals, and ASCII Policy

Scope (what SSMS covers)

- **Representation + operators only:** a portable operator canon over (m, a) with explicit guards.
- **Collapse parity:** classical results preserved under phi for every operator.
- **Streaming-safe sums:** explicit accumulators (U, W) for associative addition in batch/stream.
- **Composability:** the same operators build scalars, vectors, matrices, and polynomials.
- **Optional environment hook:** gate alignment only (a_env = g_t * a_op); magnitudes unchanged.

Non-goals (what SSMS does not do)

- **No domain physics or control:** SSMS is observation-only math, not a kinetics/controls model.
- **No operational guarantees:** decisions and safety remain with domain processes.
- **No stochastic claims:** SSMS does not add probability semantics by itself.
- **No privacy promises:** alignment is bounded but not a privacy mechanism.
- **No hidden priors:** any priors or gates must be declared explicitly in the manifest.

ASCII policy (document-wide)

- **Plain ASCII only:** formulas, identifiers, and operators use standard ASCII characters.
- **No Unicode symbols:** avoid typographic quotes, em dashes, subscripts/superscripts, or special glyphs.
- **Numeric style:** decimal point “.”, scientific notation like 1e-6, and explicit signs.
- **Consistency:** keep variable names and operator names (s_add, s_mul, etc.) consistent throughout.

SECTION 2. OPERATORS

What this section covers. A compact, delta-only canon of operators over (m, a): **s_sum, s_add/s_sub, s_mul/s_div, s_pow, s_unary, min/max/abs/round**.

Each micro-subsection follows the same 5-line pattern: **Definition (ASCII)**, **Collapse check**, **Bounds check**, **Domain guards**, **Notes** (≤2 bullets).
Streaming-safe sums use explicit accumulators **U, W**; multiplication/division use rapidity combine (**M2**). Optional gating applies **after** each operator to alignment only.

2.1 s_sum (n-ary, streaming-safe)

Definition (batch):
U = sum_i [w_i * atanh(clamp_a(a_i, eps_a))]
W = sum_i w_i
m_out = sum_i m_i
a_out = tanh(U / max(W, eps_w))

Streaming form (stateful):
U := U + sum_i [w_i * atanh(clamp_a(a_i, eps_a))]
W := W + sum_i w_i
m := m + sum_i m_i
a := tanh(U / max(W, eps_w))

Collapse check:
phi(s_sum({(m_i, a_i)})) = sum_i m_i

Bounds check:
|a_out| < 1 by clamp + tanh; W is guarded by max(W, eps_w).

Domain guards:
w_i = |m_i|^gamma >= 0; use eps_w > 0; empty set returns id_add = (0, +1).

Notes:

- Pairwise **s_add(x,y)** is the same rule applied incrementally (update U,W,m).
- Zero-magnitude results may be **displayed** as (0, +1) per zero-class display.

2.2 s_add / s_sub (pairwise, streaming-safe)

Definition (pairwise add):
w1 = |m1|^gamma ; w2 = |m2|^gamma
U = w1 * atanh(clamp_a(a1, eps_a)) + w2 * atanh(clamp_a(a2, eps_a))
W = w1 + w2
m_out = m1 + m2
a_out = tanh(U / max(W, eps_w))

Subtraction:
s_sub(x, y) = s_add(x, s_neg(y))

Collapse check:
phi(s_add((m1,a1), (m2,a2))) = m1 + m2

Bounds and guards:
|a_out| < 1 (clamp + tanh). Use max(W, eps_w). Ensure w1, w2 >= 0.

Notes:

- **Streaming:** maintain accumulators (U, W, m); adding y updates U += w2*atanh(...), W += w2, m += m2.
- **Identities:** s_add(x, id_add) = x ; s_sub(x, x) has m_out = 0 (display as (0, +1) per zero-class policy).

2.3 s_mul / s_div (M2 rapidity combine)

Helpers (alignment linearization).

```
clamp_a(a, eps_a) = sign(a) * min( abs(a), 1 - eps_a )
u(a)              = atanh( clamp_a(a, eps_a) )
```

Definition (multiplication).
Given `x1 = (m1, a1)`, `x2 = (m2, a2)`:

```
m_out = m1 * m2
a_out = tanh( u(a1) + u(a2) )

s_mul( x1, x2 ) = ( m_out , a_out )
```

Definition (division).
Magnitude follows the active division policy (strict/meadow/soft); alignment uses rapidity subtraction:

```
m_out = m1 * inv_m(m2)              # strict: 1/m2 (m2 != 0); meadow:
inv_m(0)=0; soft: guarded
a_out = tanh( u(a1) - u(a2) )

s_div( x1, x2 ) = ( m_out , a_out )
```

Collapse check (parity).

```
phi( s_mul(x1, x2) ) = m1 * m2
phi( s_div(x1, x2) ) = m1 * inv_m(m2)    # equals m1/m2 under strict with
m2 != 0
```

Bounds and guards.

```
# alignment is strictly bounded
|a_out| <= 1 - eps_a      # by clamp + tanh

# inputs are clamped before atanh to avoid singularities
a1c = clamp_a(a1, eps_a)
a2c = clamp_a(a2, eps_a)

# division definedness
#   strict: require m2 != 0
#   meadow: totalized via inv_m(0) = 0
#   soft:   denom floor via denom_soft_min > 0 (sign-preserving)
```

Identities and special cases.

```
id_mul = (1, +1)
id_add = (0, +1)

s_mul( x , id_mul ) = x
s_div( x , id_mul ) = x

# multiplying by additive identity yields zero magnitude
s_mul( x , id_add ) -> m_out = 0      # display zero-class (0, +1)

# right-inverse on magnitudes (policy-dependent)
s_mul( x , s_inv_mul(x) ) -> id_mul on magnitudes
  - holds for strict/soft, and for meadow when m_x != 0
  - if meadow and m_x == 0: m_out = 0  (recommend display (0, +1))
```

Algebraic notes (alignment path).

```
# commutative and associative under M2 combine
tanh( u(a1) + u(a2) ) = tanh( u(a2) + u(a1) )
tanh( (u(a1)+u(a2)) + u(a3) ) = tanh( u(a1) + (u(a2)+u(a3)) )
```

Tensor/broadcast convention.
For arrays/tensors, `s_mul` and `s_div` apply elementwise to `(m, a)` pairs with standard broadcasting on shapes; reductions (e.g., matmul) are defined separately via `s_einsum` and `s_sum`.

2.4 s_pow (scalar power r)

Definition (scalar r in R):
Given x = (m, a)
m_out = m^r
a_out = tanh(r * atanh(clamp_a(a, eps_a)))

Collapse check:
phi(s_pow(x, r)) = (phi(x))^r = m^r

Bounds check:
|a_out| < 1 by clamp + tanh; alignment remains bounded for any real r.

Domain guards:

- For general real r, require **m > 0**.
- If **r** is an integer, **m** may be any real; if **r < 0**, also require **m != 0** (or use **meadow_div == true** to map 0^{negative} -> 0 for magnitude).

Notes:

- **Sensitivity:** larger |r| amplifies or compresses alignment via the factor **r** in rapidity space.

- **Identity display (optional):** if **r == 0**, you may set **a_out = +1** to display exact **id_mul = (1, +1)**; the formula yields (1, 0), which is collapse-correct.

2.5 s_unary (general monotone transform f)

Definition (alignment via local log-sensitivity):
Given x = (m, a) and a monotone differentiable f, define
S_f(m) = m * f'(m) / f(m) # = d(ln|f(m)|) / d(ln m)
s_unary(f, x) = (f(m), tanh(S_f(m) * atanh(clamp_a(a, eps_a))))

Collapse check:
phi(s_unary(f, x)) = f(phi(x)) = f(m)

Bounds check:
|a_out| < 1 by clamp + tanh (for any finite S_f(m)).

Domain guards:

- f must be defined and differentiable at m, with f(m) != 0 (to avoid S_f singularities).
- Use m > 0 for functions that require it (e.g., log, power with non-integer r).
- If S_f(m) is extremely large in magnitude, optionally clip S_f(m) before use.

Notes:

- Recovers s_pow when f(m) = m^r since S_f(m) = r.
- Examples: f(m)=sqrt(m) -> S_f=1/2 ; f(m)=exp(m) -> S_f=m ; f(m)=log(m) -> S_f=1/ln(m) (use only where m>1 for stable sign).

2.6 Unified Alignment Governor (g_t): Placement & Policy

Purpose. Apply a deterministic, alignment-only "calming" gate after an operator result without touching magnitude. This preserves collapse parity and enables repeatable, observation-only control.

Canonical object.
x_op = (m_op, a_op) with m_op in R, a_op in (-1, +1).

Clamp helper (declare once).
clamp_a(a, eps_a) = sign(a) * min(abs(a), 1 - eps_a)

Gating rule (alignment-only).
m_env = m_op
a_env = clamp_a(g_t * a_op , eps_a)

Notes:

- g_t in [0, 1] by default (sign-preserving attenuation).
- Choose g_t beyond [0,1] only if you explicitly want amplification (>1) or sign inversion (<0). Default spec recommends 0 <= g_t <= 1.

Placement options.
A) Per-operator gate: apply after each SSMS verb.
B) End-of-block gate: apply once after a composed block (e.g., a matmul cell or a Horner stage).

Equivalence condition.
If g_t is constant over the block, sequential per-op gating equals one end-of-block gate with the same g_t. If g_t varies over time/index, per-op placement yields time-aware attenuation.

Composability (multiple gates).
If gates g_1, g_2, ..., g_k are applied in sequence, the effective factor is their product:
a_out = clamp_a((g_1 * g_2 * ... * g_k) * a_in , eps_a)

Guaranteed invariants.

- Collapse parity: phi(m_env, a_env) = m_env = m_op.
- Bounds: |a_env| <= 1 - eps_a (by construction).
- Zero-class display: if m_env = 0 and you adopt zero-class display, show (0, +1).

Minimal gate recipes (choose one).

- Constant: g_t = g0, where g0 in [g_min, 1].
- Exponential-calm on external stress d_t >= 0:
  g_t = max(g_min , exp(-mu * d_t))
- Logistic around threshold s_t:
  g_t = g_min + (1 - g_min) / (1 + exp(kappa * (s_t - mu)))

Parameters (recommended defaults unless the study specifies otherwise):
g_min = 0.05, mu = 0.0, kappa = 1.0.

Manifest fields (add to the single, unified manifest).
gate_used = "unified_g"
g_mode in {"constant","exp","logistic"}
g_min (float, 0 <= g_min < 1)
mu (float)
kappa (float)
eps_a (float, clamp guard, shared)

Sanity checks (per release).

- Check collapse parity on a representative op set: phi(before) == phi(after).
- Check bounds over randomized a_op and g_t draws.
- Verify composability: product-rule holds to within clamp tolerance.

Tiny examples.

- Example 1 (constant gate). eps_a=1e-6; a_op=0.80; g_t=0.50 → a_env=0.40; m unchanged.
- Example 2 (two gates). a_in=0.60; g1=0.80; g2=0.90 → a_out = clamp_a(0.72 * 0.60, eps_a) = 0.432; m unchanged.

2.7 Practical Helpers (min, max, abs, round)

Definitions (ASCII):

- **s_abs(x):** given x = (m, a)
  s_abs(x) = (|m| , a)
- **s_min(x, y):** given x = (m_x, a_x), y = (m_y, a_y)
  if m_x < m_y -> return x
  if m_y < m_x -> return y
  if m_x == m_y -> return (m_x , tanh(0.5*(atanh(clamp_a(a_x,eps_a)) + atanh(clamp_a(a_y,eps_a)))))
- **s_max(x, y):** symmetric to s_min with > in place of <
- **s_round_k(x):** round magnitude to k decimals, damp alignment by quantization gap
  m' = round(m, k)
  g_q = 1 / (1 + |m - m'| / V_unit) # clip g_q to [0,1]
  a' = g_q * a
  s_round_k(x) = (m' , a')

Collapse checks:
phi(s_abs(x)) = |m|
phi(s_min(x,y)) = min(m_x , m_y)
phi(s_max(x,y)) = max(m_x , m_y)
phi(s_round_k(x)) = round(m , k)

Bounds checks:

- s_abs passes alignment through unchanged (|a|<1 preserved).
- s_min/s_max return an existing or pooled alignment via tanh/atanh (keeps |a|<1).
- s_round_k uses 0 <= g_q <= 1, so |a'| <= |a| < 1.

Domain guards:

- For ties in s_min/s_max, you may instead use a band: if |m_x - m_y| <= tau_eq_mag, return pooled alignment; else choose the smaller/larger.
- For s_round_k, require **V_unit > 0** and integer **k >= 0**; publish **V_unit** (and optional **tau_eq_mag**) in the manifest.

Notes:

- Zero-magnitude results can be **displayed** as (0, +1) per the zero-class policy.

- The pooled tie rule keeps decisions stable without inventing new magnitudes; replace with a banded rule if preferred.

2.8 Decision Layer: Banded Comparisons (s_gt, s_eq)

Note. Use **tau_eq_mag** for the magnitude tie band on |delta_m|. If a score-based equality policy is used elsewhere, it will use **tau_eq**. Declare both only if both policies are used.

Purpose. Provide deterministic, banded decisions using classical magnitudes for the decision facet and pooled alignment for confidence. Optional gate g_t attenuates alignment only.

Weights (declare per study).
w_x = |m_x|^gamma ; w_y = |m_y|^gamma

Pooled alignment (confidence primitive).
U = w_x * atanh(clamp_a(a_x, eps_a)) + w_y * atanh(clamp_a(a_y, eps_a))
W = max(w_x + w_y , eps_w)
a_pool = tanh(U / W)
a_dec = clamp_a(g_t * a_pool , eps_a)

Comparator primitives (numeric form).
Helper: sign(z) returns -1 if z < 0, 0 if z == 0, +1 if z > 0.
Given x = (m_x, a_x), y = (m_y, a_y):
delta_m = m_x - m_y

s_gt_num(x, y) = (sign(delta_m) , a_dec)
s_eq_num(x, y, tau_eq_mag) = (1 if |delta_m| <= tau_eq_mag else 0 , a_dec)

Collapse checks.
phi(s_gt_num(x, y)) = sign(m_x - m_y)
phi(s_eq_num(x, y, tau_eq_mag)) = 1 if |m_x - m_y| <= tau_eq_mag else 0

Banded reporting (external).
Parameters: V_unit > 0 ; tau_hi >= 0 ; tau_eq_mag >= 0 ; optional tau_align in [0,1).
score_mag = |delta_m| / V_unit

Rule (four bands):

- if |delta_m| <= tau_eq_mag -> report "equal"
- else if score_mag >= tau_hi and (|a_dec| >= tau_align or tau_align not used):
  -> report "x>y" if delta_m > 0 else "x<y"
- else -> report "undecided"

Bounds & guards.

- |a_dec| < 1 by clamp + tanh; alignment gating does not touch m.
- Use eps_w > 0 to guard W; require V_unit > 0.

Notes.

- Classical facet is always decided by phi(x) vs phi(y); alignment only modulates confidence/banding.
- Set tau_eq_mag, tau_hi, (optional) tau_align in the manifest; keep g_t in [0,1] by default.

2.9 Tensor & Einstein Ops (s_einsum)

Purpose. General tensor contraction with alignment-aware semantics. Products use **M2** (rapidity add); inner sums use **U/W** pooling. Collapse parity equals the classical `einsum` on magnitudes.

Spec (Einstein form).
`s_einsum(spec, T1, T2, ..., Tk)` returns tensor `Z` with indices per `spec` (e.g., `"ik,kj->ij"`).
Each tensor element is a numeral `x = (m, a)` with `m in R`, `a in (-1, +1)`.

Helpers (alignment linearization).

```
clamp_a(a, eps_a) = sign(a) * min( abs(a), 1 - eps_a )
u(a)              = atanh( clamp_a(a, eps_a) )
```

Term composition (per contracted term).
Given a term consisting of elements `x_1, x_2, ..., x_p` along a product path:

```
m_term = prod_{r=1..p} m_r
u_term = sum_{r=1..p} u(a_r)         # rapidities add (M2)
a_term = tanh( u_term )
```

Inner sum pooling (per output index tuple).
For all terms `t` contributing to the same output position:

```
w_t   = |m_t|^gamma
U     = sum_t [ w_t * u(a_t) ]
W     = max( sum_t w_t , eps_w )
m_out = sum_t m_t
a_out = tanh( U / W )
```

Optional gate (alignment-only).

```
a_out := clamp_a( g_t * a_out , eps_a )   # post-op; m_out unchanged
```

Collapse check (parity).

```
phi( s_einsum(spec, T1, ..., Tk) ) = classical_einsum( spec, phi(T1), ...,
phi(Tk) )
```

Bounds & guards.

```
|a_out| <= 1 - eps_a          # clamp + tanh
eps_w  > 0                    # denominator guard
# shapes/index sets must be consistent with 'spec'
# if a derived contraction encodes division (rare), apply chosen
division_policy on magnitudes only
```

Streaming form (large tensors).
Maintain `(U, W, m)` per output index; update incrementally per incoming term:

```
U += w_t * u(a_t)
W += w_t
m += m_t
a  = tanh( U / max(W, eps_w) )
```

This yields batch/streaming parity on the collapsed magnitude path:

```
phi( streaming_accum ) == classical_einsum(...)   # same as batch
```

Notes.

- **Matmul** is the special case `"ik,kj->ij"` (each term is a 2-factor product, then pooled).
- Associativity on alignment arises from additivity in `u` for products and linearity of `U/W` for sums.
- Broadcasting follows the classical `einsum` rules for shapes; `(m, a)` pairs broadcast together.

2.10 Autodiff & Compositionality (u-space recipe)

Purpose. Exact, deterministic rules to propagate derivatives through SSMS ops. Magnitude follows classical calculus; alignment propagates via rapidity u = atanh(clamp_a(a, eps_a)).

Helpers (declare once).
clamp_a(a, eps_a) = sign(a) * min(abs(a), 1 - eps_a)
u = atanh(clamp_a(a, eps_a))
a = tanh(u)

Forward invariants (all ops).
• Collapse parity: phi(x) = m (unaltered by alignment path).
• Bounds: |a| < 1 by clamp + tanh.

Primitive forward rules (recap).
• s_mul(x1,x2): m' = m1*m2 ; u' = u1 + u2 ; a' = tanh(u')*
• s_div(x1,x2): m' = m1/m2 ; u' = u1 - u2 ; a' = tanh(u')
*• s_sum({x_i}): m' = sum_i m_i ; U = sum_i w_i*u_i ; W = max(sum_i w_i, eps_w) ; a' = tanh(U/W)
with w_i = |m_i|^gamma

- $s\_pow(x,r)$: $m' = m^r$; $a' = \tanh(r * u)$
- $s\_unary(f,x)$: $m' = f(m)$; $a' = \tanh(S\_f(m) * u)$ where $S\_f(m) = m * f'(m) / f(m)$

Local derivatives needed (not at clamp boundary).
dz/dm means partial derivative at fixed other inputs.
- $d(\operatorname{atanh}(a))/da = 1 / (1 - a^2)$
- $da/du = 1 - a^2$
- For $z = U/W$ with W independent of a_i: $dz/du\_i = w\_i / W$

Exact reverse-mode snippets (alignment path).
Let g_a be upstream gradient on a' (scalar).

1. $s\_mul(x1,x2)$: $a' = \tanh(u1+u2)$
   $da/da1 = (1 - a'^2) * (1 / (1 - a1^2))$
   $da/da2 = (1 - a'^2) * (1 / (1 - a2^2))$
2. $s\_div(x1,x2)$: $a' = \tanh(u1 - u2)$
   $da/da1 = (1 - a'^2) * (1 / (1 - a1^2))$
   $da/da2 = -(1 - a'^2) * (1 / (1 - a2^2))$
3. $s\_sum(\{x\_i\})$: $a' = \tanh((sum\_i\ w\_i*u\_i) / W)$ with $W = \max(sum\_i\ w\_i, eps\_w)$
   $da/da\_i = (1 - a'^2) * (w\_i / W) * (1 / (1 - a\_i^2))$
4. $s\_pow(x,r)$: $a' = \tanh(r * u)$
   $da/da = (1 - a'^2) * r / (1 - a^2)$
5. $s\_unary(f,x)$: $a' = \tanh(S\_f(m) * u)$
   $da/da = (1 - a'^2) * S\_f(m) / (1 - a^2)$
   $da/dm = (1 - a'^2) * u * dS\_f/dm$
   with $S\_f(m) = m * f'(m) / f(m)$ and
   $dS\_f/dm = f'(m)/f(m) + mf''(m)/f(m) - m(f'(m))^2/(f(m))^2$

Magnitude-path derivatives (classical).
- s_mul: $dm'/dm1 = m2$; $dm'/dm2 = m1$
- s_div: $dm'/dm1 = 1/m2$; $dm'/dm2 = -m1/(m2^2)$ (use meadow_div policy if enabled)
- s_sum: $dm'/dm\_i = 1$
- s_pow: $dm'/dm = r*m^{(r-1)}$ (require domain guards)
- s_unary: $dm'/dm = f'(m)$

Gate derivative (alignment-only).
$a\_env = clamp\_a(g\_t * a\_op , eps\_a)$
If not at clamp boundary: $da\_env/da\_op = g\_t$
At clamp boundary, use subgradient 0 (or treat as non-differentiable event in logs).
Magnitude path is unaffected by gating.

Domain guards (autodiff).
- Respect clamp boundaries: skip $1/(1 - a^2)$ where $|a| = 1 - eps\_a$; use subgradients = 0.
- Ensure $W >= eps\_w$ in s_sum; w_i depends only on m_i.
- Enforce standard domain rules for m (e.g., $m>0$ for non-integer powers or log).

Tiny worked checks.
- s_pow with $r=0.5$ (sqrt): $S\_f = 1/2$, $dS\_f/dm = 0 \Rightarrow da/dm = 0$; $da/da = (1 - a'^2) * (1/2) / (1 - a^2)$.
- s_unary with $f=\exp$: $S\_f = m$; $dS\_f/dm = 1 \Rightarrow da/dm = (1 - a'^2) * u$.

Notes.
• These rules are exact under the SSMS definitions (no approximations).
• For tensors, apply the same derivatives elementwise; s_einsum uses the s_mul and s_sum rules above.

2.11 Performance Modes (big-data safe, exact-by-default)

Policy. Exact math is the default. Approximations are opt-in via manifest flags and must never break collapse parity. Always pre-clamp alignment before any transform.

Clamp helper (declare once).
clamp_a(a, eps_a) = sign(a) * min(abs(a), 1 - eps_a)

Keep u-space internal (reduce calls).

- Cache u_i = atanh(clamp_a(a_i, eps_a)) once per input element.
- In streaming s_sum/s_einsum, maintain (U, W, m) and compute a_out only at checkpoints:
  a_out = tanh(U / max(W, eps_w))

Approximation switches (manifest).
approx_mode in {"exact","poly","rational","fast_sat"}
approx_atanh_domain = a_poly_max # e.g., 0.8
approx_tanh_domain = u_rational_max # e.g., 3.0
target_abs_err = 1e-3 # implementation goal; verify in QA

atanh approximations (choose one; pre-clamp first).

1. poly (good for |a| <= a_poly_max):
   atanh_approx(a) = a + (a^3)/3 + (a^5)/5
   fallback to exact atanh(a) outside the domain.
2. rational (Padé-style on |a| <= a_poly_max):
   atanh_approx(a) = a * (1 + (a^2)/3 + (a^4)/5)
   fallback to exact atanh(a) outside the domain.

tanh approximations (choose one; bounded by design).

1. rational (good for |u| <= u_rational_max):
   tanh_approx(u) = u * (27 + u^2) / (27 + 9*u^2)
   fallback to exact tanh(u) outside the domain.
2. fast_sat (saturation branch):
   if |u| >= u_sat then return sign(u) * (1 - eps_a) else return tanh_approx(u)
   Note: choose u_sat to meet target_abs_err; clamp final result.

Exact fallbacks (always available).
atanh_exact(a) = 0.5 * ln((1 + a) / (1 - a))
tanh_exact(u) = (e^(u) - e^(-u)) / (e^(u) + e^(-u))

Safety invariants (must hold in all modes).

- Collapse parity: phi is always computed from m only; unaffected by approximations.
- Bounds: apply clamp_a to every alignment input and to the final a_out.
- Streaming: W is guarded by max(W, eps_w).
- Determinism: same inputs + same approx_mode -> identical outputs.

QA hooks (minimal).

- Uniform grid on a in (-1+eps_a, 1-eps_a): check |atanh_approx(a) - atanh_exact(a)| <= target_abs_err inside domain.
- Uniform grid on u in [-u_rational_max, +u_rational_max]: check |tanh_approx(u) - tanh_exact(u)| <= target_abs_err inside domain.
- End-to-end: s_sum, s_mul, s_pow, and a small s_einsum run with approx_mode vs exact must match within target_abs_err on alignment and exactly on collapse.

Manifest fields (add to unified manifest).
approx_mode, a_poly_max, u_rational_max, u_sat, target_abs_err

Tiny example (illustrative).

- Given a = 0.40, eps_a = 1e-6, poly mode:
  atanh_approx(0.40) = 0.40 + 0.40^3/3 + 0.40^5/5 = 0.4237 (approx)
  exact atanh(0.40) = 0.4236 (close; within 1e-3 target).
- Given u = 0.85, rational mode:
  tanh_approx(0.85) = 0.85*(27+0.85^2)/(27+9*0.85^2) ≈ 0.6897
  exact tanh(0.85) ≈ 0.6900 (close; within 1e-3 target).

2.12 Linear Algebra Stability Notes (A_beta bound)

Purpose. Provide a simple, deterministic bound on alignment growth across linear ops (matvec/matmul) under SSMS semantics. Magnitudes follow classical linear algebra; alignment uses product-as-rapidity-add and sum-as-U/W pooling.

Setup (row i for y = A x).
Let a_ij = (m_ij, a_ij) and x_j = (m_j, a_j).
u_ij = atanh(clamp_a(a_ij, eps_a))
u_xj = atanh(clamp_a(a_j, eps_a))
Term product for y_i uses M2: u_term(ij) = u_ij + u_xj.
Weights per contracted term: w_ij = | m_ij * m_j |^gamma
Row pool (before any gate):
U_i = sum_j [w_ij * u_term(ij)]
W_i = max(sum_j w_ij , eps_w)
u_yi = U_i / W_i
a_yi = tanh(u_yi)

Row-average decomposition.
Define weighted row averages

avg_uA_i = (sum_j w_ij * u_ij) / W_i
avg_uX_i = (sum_j w_ij * u_xj) / W_i
Then u_yi = avg_uA_i + avg_uX_i.

Row stability indicator.
A_beta_row(i) = | avg_uA_i |
A_beta = max_i A_beta_row(i) # layer scalar

Single-layer bound (no gate).
Using |avg_uX_i| <= max_j |u_xj|, we have
|u_yi| <= A_beta_row(i) + ||u_x||_inf
Thus
||u_y||_inf <= A_beta + ||u_x||_inf

With gate (alignment-only).
If a per-row (or global) gate g_t is applied after the row pool:
u_yi_env = g_t * u_yi (before tanh and clamp)
Then
||u_y_env||_inf <= |g_t| * (A_beta + ||u_x||_inf)

Multi-layer stack (L layers, gates g_1..g_L).
Let the l-th layer have A_beta^(l) and gate g_l. The recurrence is
U_{l+1} <= |g_l| * (A_beta^(l) + U_l) where U_l := ||u^{(l)}||*inf*
Solving, with U_0 := ||u_in||inf:
*U_L <= (prod{k=1..L} |g_k|) * U_0*
+ *sum*{s=1..L} (A_beta^(s) * prod_{k=s..L} |g_k|)

Practical recipe.
• Compute u_ij once (cached). Track A_beta per layer at checkpoints.
• If U_L exceeds a declared design ceiling U_ceiling, tighten gate(s): choose g_l in [0,1] to meet the bound.
• Publish A_beta and chosen gates in the manifest for reproducibility.

Invariants.
• Collapse parity holds: phi(y) = classical(A) * phi(x).
• Bounds preserved by final tanh + clamp_a.
• A_beta is data-dependent (depends on w_ij which depend on magnitudes).

Manifest fields (optional).
A_beta_target (float), U_ceiling (float), gate_policy = "row" or "global"

Tiny check (illustrative).
If A_beta = 0.30, ||u_x||_inf = 0.40, g_t = 0.80, then
||u_y_env||_inf <= 0.80 * (0.30 + 0.40) = 0.56.

2.13 Division Policy & Edge Cases (meadow_div)

Purpose. Make division a total, deterministic operator on **magnitudes** while preserving the bounded **alignment** semantics. Default is **strict** (classical) division; optional `meadow_div` enables a **total inverse** on the magnitude path. A "soft" guard policy is provided for numerical robustness.

Helpers (declare once).

```
clamp_a(a, eps_a) = sign(a) * min( abs(a), 1 - eps_a )
u(a)              = atanh( clamp_a(a, eps_a) )     # linearized stability
(rapidity)
```

Operators (strict by default).

- **Reciprocal (multiplicative inverse).**

```
s_inv_mul( (m, a) ) = ( 1/m , tanh( -u(a) ) )   # domain: m != 0
```

- **Division.**

```
s_div( (m1, a1), (m2, a2) ) = ( m1/m2 , tanh( u(a1) - u(a2) ) )   # domain:
m2 != 0
```

Optional magnitude policies (choose one via manifest).

- **division_policy = "strict"** (default): require `m2 != 0` (domain error otherwise).
- **division_policy = "meadow"** (totalized inverse on magnitudes; `meadow_div == true` is a synonym):

```
inv_m(m) = 0 if m == 0 else 1/m
# then
s_inv_mul_magnitude = inv_m(m)
s_div_magnitude     = m1 * inv_m(m2)
```

- **division_policy = "soft"** (guard tiny denominators; preserve sign):

```
sign_star(x) = +1 if x >= 0 else -1      # define sign(0) := +1 to avoid 0-
denominator
denom        = sign_star(m2) * max( |m2| , denom_soft_min )
s_div_magnitude = m1 / denom
```

Alignment path (independent of policy).

```
# reciprocal
a_out = tanh( -u(a) )                    # equals -a in exact math; clamp_a
enforces bounds
# division
a_out = tanh( u(a1) - u(a2) )
# optional environment gate (alignment-only, after op)
a_out := clamp_a( g_t * a_out , eps_a )
```

The meadow/soft choices affect **only** the magnitude path; alignment semantics are unchanged. Collapse parity remains `phi(out) = m_out` by definition.

Collapse checks.

```
# strict
phi( s_div(x1, x2) ) = m1 / m2                    # defined only if m2 != 0
# meadow
phi( s_div(x1, x2) ) = m1 * inv_m(m2)             # defined for all m2;
equals 0 when m2 == 0
# soft
phi( s_div(x1, x2) ) = m1 / denom                 # with denom as defined
above
```

Bounds & guards.

```
|a_out| <= 1 - eps_a     # clamp + tanh
# magnitude safety
meadow/soft: no NaN/Inf on magnitude path
soft: choose denom_soft_min > 0 and publish it
```

Identities & edge cases.

- **Right-inverse on magnitudes (non-meadow).**

```
s_mul( x , s_inv_mul(x) )  ->  magnitude 1  (if division_policy !=
"meadow")
# alignment combines to tanh( u(a) + (-u(a)) ) = 0; identity is on
magnitudes
```

- **Meadow x==0.** If `meadow_div` and `m == 0`:

```
s_mul( x , s_inv_mul(x) )  ->  m_out = 0        # collapses to 0; recommend
zero-class display (0, +1)
```

- **Multiplicative identity.**

```
id_mul = (1, +1)
s_div( x , id_mul ) = x     # for all policies (strict/meadow/soft)
```

- **Additive identity in denominator.**

```
id_add = (0, +1)
# strict: undefined
# meadow: m_out = m1 * inv_m(0) = 0
# soft:   m_out = m1 / ( +1 * denom_soft_min )     # finite, sign-
preserving
```

Manifest fields (unified manifest additions).

```
division_policy  in {"strict", "meadow", "soft"}
meadow_div       (bool; synonym for division_policy == "meadow")
denom_soft_min   (float > 0; required if division_policy == "soft")
```

Tiny examples (illustrative).

- **Strict.**

```
x1 = (6, 0.40),  x2 = (2, 0.10)
m_out = 6/2 = 3
a_out = tanh( u(0.40) - u(0.10) )
```

- **Meadow (m2 = 0).**

```
x1 = (5, 0.40),  x2 = (0, 0.20)
m_out = 5 * inv_m(0) = 0
a_out = tanh( u(0.40) - u(0.20) )
# display (0, +1) if you adopt the zero-class policy
```

- **Soft (tiny denominator).**

```
denom_soft_min = 1e-6
# let |m2| << 1e-6 and m2 >= 0
denom = +1 * 1e-6
m_out = m1 / 1e-6            # finite; alignment unchanged by the guard
```

Minimal QA invariants (L0–L1 for division).

```
# L0 collapse parity (per policy)
assert phi( s_div(x1, x2) ) == classical_div_policy(phi(x1), phi(x2))
# L0 alignment bounds
assert |a_out| <= 1 - eps_a
# L1 meadow totality
if division_policy == "meadow":  assert is_finite( phi( s_div(x1, x2) ) )
# all m2
# L1 soft floor
if division_policy == "soft":    assert |phi( s_div(x1, x2) )| <= |m1| /
denom_soft_min + tiny_tol
```

2.14 Polynomials & Horner Bridge (s_horner)

Purpose. Evaluate polynomials with SSMS semantics using Horner's method. Products use M2 (rapidity add); adds use U/W pooling. Collapse parity equals classical Horner on magnitudes.

Definition (Horner form).
Given polynomial p(t) = c_0 + c_1 t + ... + c_n t^n and x = (m_x, a_x).
Let c_k be numerals c_k = (m_ck, a_ck).
Initialize: y := c_n
For k = n-1 down to 0:
y := s_add(s_mul(y , x) , c_k)
Return y_out := y
Optional gate (alignment-only) can be applied after each step or once at the end:
y_out := (m_y , clamp_a(g_t * a_y , eps_a))

Collapse check.
phi(s_horner(p, x)) = classical_horner(phi(p), phi(x))

Bounds check.
All intermediate alignments remain in (-1, +1) by clamp + tanh in s_mul and s_add.

Domain guards.

- Ensure each c_k is a valid numeral; no special domain needed beyond existing s_mul/s_add requirements.
- For large n, prefer streaming: compute a_out only at checkpoints; maintain cached u values for inputs reused across steps.

Notes.

- s_pow(x, r) for integer r can be synthesized via Horner on monomials if desired, but native s_pow is preferred.
- When coefficients are classical scalars (zero-class display), use c_k = (m_ck, +1).

Tiny example (illustrative).
Let p(t) = 2 + 3 t + 1 t^2 (n=2).
Coefficients: c_2=(1,+1), c_1=(3,+1), c_0=(2,+1).
Input: x=(m_x, a_x).
Step 1: y := c_2 = (1,+1)
Step 2: y := s_add(s_mul(y , x) , c_1)
Step 3: y := s_add(s_mul(y , x) , c_0)
Collapse parity: phi(y) = 2 + 3m_x + 1m_x^2 (classical), while alignment is composed via M2 (products) and U/W pooling (adds).

2.15 Error Budget Channel (s_error)

Purpose. Apply a deterministic, alignment-only attenuation to reflect implementation/numeric error budgets without introducing probability semantics. This is a named helper equivalent to a gate with a computed factor g_err in [0,1].

Definition.
Given x = (m, a), err_max >= 0 (scalar error budget), and lambda_e >= 0 (sensitivity):
g_err = max(g_min , exp(-lambda_e * err_max))
s_error(x , err_max) = (m , clamp_a(g_err * a , eps_a))

Collapse check.
phi(s_error(x, err_max)) = m

Bounds & guards.

- 0 <= g_err <= 1 ensures |a_out| <= |a| < 1 (after clamp).
- Choose g_min in [0,1); set g_min > 0 if you need a floor against total collapse.
- err_max is a declared bound (units documented in the manifest).

Composition (multiple errors).
For errors e_1, e_2, ..., e_k with the same lambda_e:
g_total = prod_i exp(-lambda_e * e_i) = exp(-lambda_e * sum_i e_i)
Thus s_error(... e_1) then s_error(... e_2) equals one s_error with err_sum = sum_i e_i.

Relation to unified gate.
s_error is equivalent to applying the unified gate with g_t := g_err at the chosen placement (per-op or end-of-block). Magnitude is never altered.

Manifest fields (add to unified manifest).
error_channel_used = true/false
lambda_e (float >= 0)
g_min (float in [0,1))
error_metric (string, e.g., "ulp_bound", "abs_err", "rel_err")
error_unit (string, e.g., "ulp", "abs", "rel")

Tiny examples (illustrative).

- Example 1: a=0.70, err_max=0.05, lambda_e=4.0, g_min=0.05 → g_err=exp(-0.20)=0.8187 → a_out=0.5731 (then clamp).
- Example 2 (two steps): e1=0.02, e2=0.03, lambda_e=5.0 → g_total=exp(-5*(0.05))=exp(-0.25)=0.7788 → a shrinks by the same factor in one shot.

2.16 Identities, Neutral & Absorbing Elements (quick canon)

Purpose. Fix the neutral/identity elements and inverses for SSMS operators; clarify zero-class display and collapse parity.

Canonical identities.

```
id_add = ( 0 , +1 )
id_mul = ( 1 , +1 )
```

Additive inverse.
Flips magnitude; preserves alignment.

```
s_neg( (m, a) ) = ( -m , a )
# weight w = |m|^gamma is unchanged by sign flip; alignment is not negated
```

Multiplicative inverse (recap).

```
clamp_a(a, eps_a) = sign(a) * min( abs(a), 1 - eps_a )
u(a)              = atanh( clamp_a(a, eps_a) )

s_inv_mul( (m, a) ) = ( inv_m(m) , tanh( -u(a) ) )
```

Magnitude path `inv_m(m)` follows the chosen division policy:

```
# strict:   inv_m(m) = 1/m           (domain m != 0)
```

```
# meadow:   inv_m(0) = 0 ; otherwise 1/m   (totalized)
# soft:     inv_m(m) = 1 / ( sign_star(m) * max(|m|, denom_soft_min) )
```

Absorbing & neutral behavior.

```
s_add( x , id_add ) = x
s_mul( x , id_mul ) = x
s_mul( x , id_add ) -> magnitude 0  # absorbing; display (0, +1) if zero-
class display is enabled
```

Zero-class display (deterministic).

```
if m_out == 0: display ( 0 , +1 )   # regardless of incoming alignment
# Rationale: avoid inventing directional bias at exact-zero magnitude
```

Distributivity & collapse parity.
Collapse layer is classical:

```
phi( s_mul( x , s_add(y,z) ) )            = phi(x) * ( phi(y) + phi(z) )
phi( s_add( s_mul(x,y) , s_mul(x,z) ) )   = phi(x)*phi(y) + phi(x)*phi(z)
```

Alignment layer composes via rapidity add (products) and U/W pooling (sums); no extra law is imposed beyond operator definitions. Always report classical results via `phi`, alignment as confidence.

Idempotents (helpers).

```
s_min(x, x)           = x
s_max(x, x)           = x
s_abs( s_abs(x) )     = s_abs(x)                 # by definition of s_abs
s_round_k( s_round_k(x) ) = s_round_k(x)         # same k and V_unit
```

Bounds & guards.

```
# alignment pre-clamp
a' = clamp_a(a, eps_a)

# pooling guard
W = max( W , eps_w )

# inverses/division respect policy
division_policy in {"strict","meadow","soft"}
```

Tiny checks (illustrative).

```
s_add( (2, 0.4) , id_add ) = (2, 0.4)        # under streaming U/W
s_mul( (5, 0.3) , id_mul ) = (5, 0.3)
s_mul( (7, 0.2) , id_add ) -> (0, +1)        # if zero-class display is
enabled
```

2.17 Physics Micro-Examples

Helpers (declare once).
clamp_a(a, eps_a) = sign(a) * min(abs(a), 1 - eps_a)
u(a) = atanh(clamp_a(a, eps_a))
Defaults for examples: eps_a = 1e-6 ; eps_w = 1e-12 ; gamma = 0 ; g_t = 1 unless stated.

A) Spring force with damping gate (F = k * x)
Inputs: k = (m_k, a_k), x = (m_x, a_x).
Numeral product: F_num = s_mul(k , x)

m_F = m_k * m_x
a_F_raw = tanh(u(a_k) + u(a_x))
Optional gate (alignment-only): a_F = clamp_a(g_t * a_F_raw , eps_a)

Collapse check: phi(F_num) = m_F.

Tiny numbers (illustrative):
m_k = 100.0 ; a_k = 0.90
m_x = 0.05 ; a_x = 0.60
u(a_k) ~ 1.4722 ; u(a_x) ~ 0.6931 -> a_F_raw ~ tanh(2.1653) ~ 0.9734
m_F = 5.0
If g_t = exp(-0.6) ~ 0.5488 -> a_F ~ 0.534 (then clamp).

B) Kinetic energy via Horner/pow (E = 0.5 * m * v^2)
Inputs: m = (m_m, a_m), v = (m_v, a_v), c = (0.5, +1).
Step 1: v2 = s_pow(v , 2)
m_v2 = (m_v)^2
a_v2 = tanh(2 * u(a_v))
Step 2: t = s_mul(m , v2)
m_t = m_m * m_v2
a_t = tanh(u(a_m) + u(a_v2))
Step 3: E_num = s_mul(c , t)
m_E = 0.5 * m_t
a_E = tanh(u(+1) + u(a_t)) = a_t

Collapse check: phi(E_num) = 0.5 * m_m * (m_v)^2.

Tiny numbers (illustrative):
m_m = 2.0 ; a_m = 0.20 -> u(a_m) ~ 0.2027
m_v = 3.0 ; a_v = 0.60 -> u(a_v) ~ 0.6931 ; a_v2 = tanh(1.3863) ~ 0.8820 ; u(a_v2) ~ 1.3926
u(a_t) ~ 0.2027 + 1.3926 = 1.5953 -> a_t ~ tanh(1.5953) ~ 0.9213
m_E = 0.5 * 2.0 * 9.0 = 9.0

2.18 Finance Micro-Examples

Helpers (declare once).
clamp_a(a, eps_a) = sign(a) * min(abs(a), 1 - eps_a)
u(a) = atanh(clamp_a(a, eps_a))
Defaults for examples: eps_a = 1e-6 ; eps_w = 1e-12 ; gamma = 0 ; g_t = 1 unless stated.

A) Portfolio P/L with banded decision
Two legs: x1 = (m1, a1), x2 = (m2, a2).
Portfolio: P = s_sum({x1, x2})

w1 = w2 = 1
U = u(a1) + u(a2)
W = max(2 , eps_w) = 2
m_P = m1 + m2
a_P = tanh(U / W)
Decision vs zero using Section 2.8 banding.

Tiny numbers (illustrative):
m1 = +1200 ; a1 = 0.70 -> u(a1) ~ 0.8673
m2 = -800 ; a2 = -0.20 -> u(a2) ~ -0.2027
U ~ 0.6646 ; a_P ~ tanh(0.3323) ~ 0.3204 ; m_P = +400
With V_unit = 100 ; tau_eq_mag = 25 ; tau_hi = 3.0 -> score_mag = 4.0 -> report "x>y" (positive P/L).

B) Risk-adjusted return (return divided by volatility)
Return R = (P_end - P_start) as numeral r = (m_r, a_r).
Volatility sigma as numeral s = (m_s, a_s).
Risk-adjusted score q = s_div(r , s)

m_q = m_r / m_s (or soft/meadow per 2.13)
a_q = tanh(u(a_r) - u(a_s))
Optional gate: a_q := clamp_a(g_t * a_q , eps_a)

Collapse check: phi(q) = m_r / m_s.

Tiny numbers (illustrative):
m_r = 0.08 ; a_r = 0.50 -> u(a_r) ~ 0.5493
m_s = 0.04 ; a_s = 0.10 -> u(a_s) ~ 0.1003
m_q = 2.0 ; a_q ~ tanh(0.5493 - 0.1003 = 0.4490) ~ 0.4217

2.19 Telecom & Signals Micro-Examples

Helpers (declare once).
clamp_a(a, eps_a) = sign(a) * min(abs(a), 1 - eps_a)
u(a) = atanh(clamp_a(a, eps_a))
Defaults for examples: eps_a = 1e-6 ; eps_w = 1e-12 ; gamma = 0 ; g_t = 1 unless stated.

A) Streaming mean one-way delay (ms) with stability

Goal: maintain a running mean delay where alignment reflects stability of recent samples.

Inputs: per-packet delays d_i as numerals x_i = (m_i, a_i).
Streaming s_sum state (per window): (U, W, m) with w_i = 1.

Update per batch (or per packet):
U := U + sum_i [atanh(clamp_a(a_i, eps_a))]
W := W + count_i
m := m + sum_i m_i
a_pool := tanh(U / max(W, eps_w))

Mean as a unary scale (keeps alignment):
Let K be packet count so far.
y_sum = (m , a_pool)
mean = s_unary(f, y_sum) with f(m) = m / K
m_mean = m / K
a_mean = tanh(S_f * atanh(clamp_a(a_pool, eps_a))) with S_f = 1
Thus a_mean = a_pool.

Tiny numbers (illustrative):
Samples: (40, 0.70), (60, 0.50), (50, 0.80)
U ≈ 2.5152 ; W = 3 → a_pool ≈ tanh(0.8384) ≈ 0.6850
m = 150 → m_mean = 50.0 ; a_mean = 0.6850
Interpretation: classical mean 50.0 ms; alignment 0.685 indicates reasonably stable delays.

B) Signal-to-noise quality (SNR) with banded decision

Define SNR numeral q = s_div(S , N), where S = (m_S, a_S) is signal power and N = (m_N, a_N) is noise power.

Magnitude and alignment:
m_q = m_S / m_N # or per 2.13 policy
a_q = tanh(u(a_S) - u(a_N))
Optional gate (alignment-only): a_q := clamp_a(g_t * a_q , eps_a)

Decision vs threshold T (classical):
delta_m = m_q - T
Use Section 2.8 banding to report {x>y, x<y, equal, undecided}.

Tiny numbers (illustrative):
S = (20, 0.60) ; N = (5, 0.20)
m_q = 4.0
u(a_S) ≈ 0.6931 ; u(a_N) ≈ 0.2027 → a_q ≈ tanh(0.4904) ≈ 0.4545
With T = 3.0 and V_unit = 1, tau_hi = 3.0: score_mag = |4.0 - 3.0| / 1 = 1.0

Report "x>y" (quality above threshold). Alignment ≈ 0.4545 conveys moderate confidence; g_t can attenuate it without changing 4.0.

2.20 Universality Note — SSM + SSMS Complement Core Mathematics (applies to every domain)

Purpose. State clearly that SSM (numerals) and SSMS (operators) **extend, not replace** classical math. Every classical formula lifts verbatim; collapse parity guarantees the same numeric results, while alignment adds a bounded stability/confidence signal.

Lift rule (any classical f).
Given classical inputs `m` (scalars, vectors, tensors), define `x = (m, a)` with `a in (-1, +1)`. Let `F` be the SSMS expression obtained by replacing classical ops with SSMS verbs. Then:

```
phi( F( (m_1,a_1), (m_2,a_2), ... ) ) = f( m_1, m_2, ... )
```

This is the **collapse homomorphism**: classical on magnitudes; alignment is carried alongside.

Gate neutrality (observation-only).

```
a_env = clamp_a( g_t * a_op , eps_a )
phi( (m_op, a_env) ) = m_op
```

Gates attenuate alignment only; magnitudes (and collapse) are unchanged.

Zero-class display (deterministic).

```
if m == 0: display (0, +1)
```

No directional bias is invented at exact zero.

Why this means "every domain."

- Physics, finance, telecom, chemistry, biology, imaging, geospatial, governance, healthcare, AI/ML, climate, weather, cybersecurity, and more.
- Replace classical verbs with SSMS verbs, keeping units and models intact:
  - `+ , - , * , / , ^` -> `s_add , s_neg , s_mul , s_div , s_pow`
  - `sum/reduce` -> `s_sum` (U/W pooling)
  - `matmul/einsum` -> `s_einsum` (M2 for products + U/W for sums)
  - `min/max` -> `s_min / s_max`
  - comparisons -> `s_gt / s_eq` + banded reporting
- Classical outputs remain identical via `phi`; alignment exposes stability/drift for safer judgment under uncertainty.

Minimal migration checklist.

1. **Wrap values.** Represent each measured/computed value as `x = (m, a)`; choose `gamma >= 0`, `eps_a` (e.g., `1e-6`), `eps_w` (e.g., `1e-12`).
2. **Swap verbs.** Replace classical operators with SSMS verbs as listed above; keep the original equations and units.
3. **Division policy.** Pick one:
4. `division_policy in {"strict","meadow","soft"}`
5. `# if "soft": choose denom_soft_min > 0`
6. **Optional gate.** Decide a fixed release policy:
7. `gate_used in {"on","off"} ; if "on": publish lane_recipe, kappa>=0, mu>=0, rho in [0,1], Q0 in [0,1]`
8. **Bands for decisions.** Calibrate once, then freeze:
9. `V_eps > 0, beta_v >= 0, lambda_u >= 0`
10. `tau_hi in (0,1), tau_eq in (0,1)`
11. `alpha_gt in (0,1), alpha_eq in (0,1), ref_size, split_seed, policy_gate in {"on","off"}`
12. **Publish a tiny manifest.** Include at least:
13. `gamma, eps_a, eps_w,`
14. `division_policy (and denom_soft_min if "soft"),`
15. `gate_used (+ knobs if "on"),`
16. `V_eps, beta_v, lambda_u, tau_hi, tau_eq,`
17. `alpha_gt, alpha_eq, ref_size, split_seed, zero_class_display`
18. **Report.** Emit **classical outputs via phi** (numbers users expect) **plus alignment** for confidence and banded reporting.

One-line takeaway.
"Write the same equations, get the same numbers, now with a trustworthy stability signal."

SECTION 3. COMPARISONS & BANDS

What this section covers. Numeric comparison operators that produce **scores** (not booleans), plus **band thresholds** for reporting **true / false / undecided** (and **equal / anti-equal / uncertain**). The scores are alignment-aware, observation-only, and **collapse-safe** (collapse returns classical comparisons). Optional gating may be applied **after** scoring to attenuate alignment only.

3.1 s_gt (greater-than score)

Definition (score in (-1, +1)):
Given $x=(m_x,a_x)$, $y=(m_y,a_y)$

$$dv = (m_x - m_y) / \max(|m_x| + |m_y| , V_{eps})$$

$$du = \operatorname{atanh}(\operatorname{clamp\_a}(a_x, eps_a)) - \operatorname{atanh}(\operatorname{clamp\_a}(a_y, eps_a))$$

$$s_{gt}(x,y) = \tanh(beta_v * dv + lambda_u * du)$$

Interpretation:
s_gt > 0 => “x > y” tendency
s_gt < 0 => “x < y” tendency
s_gt ~ 0 => tie/uncertain (use bands)

Collapse check:
If lambda_u = 0 (or a_x = a_y), sign(s_gt) = sign(m_x - m_y) after normalization; i.e., reduces to the classical comparison.

Bounds and guards:

- s_gt in (-1, +1) by tanh.
- Choose V_eps > 0, beta_v >= 0, lambda_u >= 0.
- Clamp alignments before atanh to avoid singularities.

Reporting bands (publish once):
If s_gt >= tau_hi -> report “true (x > y)”;
If s_gt <= -tau_hi -> report “false (x > y)”;
Else -> “undecided”.

3.2 s_eq (equality / near-equality score)

Definition (score in (-1, +1]):
Given x=(m_x,a_x), y=(m_y,a_y)
dv = (m_x - m_y) / max(|m_x| + |m_y| , V_eps)
u_x = atanh(clamp_a(a_x, eps_a))
u_y = atanh(clamp_a(a_y, eps_a))
d_u = | u_x - u_y |
raw = beta_v * abs(dv) + beta_u * d_u
s_eq(x,y) = 1 - 2 * tanh(raw)

Interpretation:
s_eq ~ +1 => “equal / very close”
s_eq ~ 0 => “similar / modest difference”
s_eq ~ −1 => “anti-equal / very different”

3.3 Banded Reporting Policy (global)

Purpose. Turn numeric comparison scores into stable, human-readable outcomes.

Thresholds (declare once).

```
tau_hi in (0,1)   # band for s_gt (">" / "<")
tau_eq in (0,1)   # band for s_eq ("equal")
```

Mapping (precedence: equality first).

```
if s_eq(x,y) >= tau_eq:          report "equal"
else if s_gt(x,y) >= +tau_hi:    report "x > y"
else if s_gt(x,y) <= -tau_hi:    report "x < y"
else:                            report "undecided"
```

Consistency requirements.

```
# use the SAME numeric knobs and gate state everywhere:
V_eps, eps_a, eps_w    # guards
beta_v, lambda_u       # s_gt weights
gate_used in {"on","off"}  # must match calibration (see §3.4)
```

Gate effect (optional, alignment-only).

```
a_env = clamp_a( g_t * a_op , eps_a )    # post-op
# shrinking |du| via gating shifts reports to rely more on magnitude
differences,
# but collapse-level comparisons remain unchanged.
```

Publishing (manifest).

```
tau_hi, tau_eq                 # final bands
alpha_gt, alpha_eq             # chosen risks (from §3.4)
ref_size, split_seed           # calibration bookkeeping
policy_gate in {"on","off"}    # gate state used during calibration
```

Tiny examples (illustrative, using calibrated bands).

```
Assume:
  tau_hi = 0.69
  tau_eq = 0.81

Case A:
  s_gt(x,y) = +0.73,  s_eq(x,y) = 0.40  ->  "x > y"   (since +0.73 >= 0.69)

Case B:
  s_gt(x,y) = -0.72,  s_eq(x,y) = 0.35  ->  "x < y"   (since -0.72 <= -
0.69)

Case C:
  s_gt(x,y) = +0.55,  s_eq(x,y) = 0.85  ->  "equal"   (equality takes
precedence)

Case D:
  s_gt(x,y) = +0.20,  s_eq(x,y) = 0.60  ->  "undecided"
```

Minimal QA (L0–L1).

```
# L0 determinism
re-run with same inputs and manifest -> identical reports

# L1 precedence safety
if s_eq >= tau_eq: report must be "equal" regardless of s_gt
```

```
# L1 symmetry sanity
report(x,y) == invert(report(y,x)) for ">" / "<"; "equal" and "undecided"
are symmetric
```

3.4 Band Calibration (global, minimal and reproducible)

Purpose. Provide a tiny, deterministic protocol to pick `tau_hi` (for `s_gt`) and `tau_eq` (for `s_eq`) from a reference set, with bounded error and no diagrams. The bands become part of the manifest and must be frozen for a release.

Inputs (declare once).

```
D_ref = { (x_i, y_i, label_i) } ,   label_i in { ">", "<", "=" }
policy_gate  in { on, off }   # use the SAME gate choice here as in
deployment
V_eps, eps_a, eps_w           # same numeric guards as in §3.3
alpha_gt in (0,1)             # tail risk for ">" / "<"
alpha_eq in (0,1)             # tail risk for "="
```

Scoring pass (single sweep).
For each `(x,y,label)` in `D_ref`, compute

```
g  = s_gt(x, y)        # antisymmetric score in (-1, +1)
e  = s_eq(x, y)        # symmetric score in (-1, +1)
```

Use the same `policy_gate` (on/off) and the same `V_eps, eps_a, eps_w` here and in production. Build three lists:

```
Gpos = { g | label == ">" }
Gneg = { -g | label == "<" }      # flip sign so both are "evidence for
strong ordering"
Eq   = { e | label == "=" }
```

Calibration (quantile, risk-controlled).
Choose bands by upper quantiles on evidence-for-truth distributions:

```
tau_hi = min( Q_{1 - alpha_gt}(Gpos) , Q_{1 - alpha_gt}(Gneg) )
tau_eq =      Q_{1 - alpha_eq}(Eq)
```

where `Q_p(S)` is the p-quantile of set `S`. This yields a **single** `tau_hi` that is symmetric for ">" and "<", and an equality band `tau_eq` that reflects how strict you are about declaring equality.

Recommended defaults (practical).

```
alpha_gt = 0.05     # controls false ">" or "<" on labeled opposites
alpha_eq = 0.10     # controls false "equal" on non-equal pairs
```

Tighter bands (smaller alphas) reduce false decisions but increase "undecided."

Mapping (uses §3.3 as-is).

```
if e >= tau_eq                  -> "equal"
else if g >= +tau_hi            -> "x > y"
else if g <= -tau_hi            -> "x < y"
else                            -> "undecided"
```

The precedence of "equal" before "greater/less" is mandatory to prevent contradictory reports.

Cross-checks (consistency and symmetry).

```
# C1: symmetry of ">" / "<"
assert tau_hi == min( Q_{1 - alpha_gt}(Gpos), Q_{1 - alpha_gt}(Gneg) )

# C2: range sanity
assert 0 < tau_hi < 1
assert 0 < tau_eq < 1

# C3: gating consistency
# Calibrate and deploy with the same gate state (on/off). Do not mix.
```

Holdout validation (L2).
Split `D_ref` into `train` and `holdout` (e.g., 80/20, stratified by label). Calibrate on `train`, then report on `holdout`:

```
err_gt  = P_holdout( report is ">" but label != ">" ) + P_holdout( report
is "<" but label != "<" )
err_eq  = P_holdout( report is "equal" but label != "=" )
u_rate  = P_holdout( report is "undecided" )
```

Publish `(tau_hi, tau_eq, err_gt, err_eq, u_rate, |D_ref|, split_seed)` in the manifest.

Minimal streaming sanity.
Because `s_gt` uses a normalized magnitude difference and `s_eq` is symmetric, calibration is stable under scaling of magnitudes. Still verify:

```
scale > 0:
assert report( scale*x, scale*y ) == report( x, y )
```

Manifest additions.

```
bands:
  policy_gate: "on" | "off"
  tau_hi: <float in (0,1)>
  tau_eq: <float in (0,1)>
  alpha_gt: <float in (0,1)>
  alpha_eq: <float in (0,1)>
  ref_size: <int>
  split_seed: <int>      # for reproducibility of train/holdout
```

Tiny worked example (illustrative).

```
Given (alpha_gt, alpha_eq) = (0.05, 0.10)
```

```
Compute Gpos, Gneg, Eq on D_ref (gate=off).
Suppose:
  Q_0.95(Gpos) = 0.72,  Q_0.95(Gneg) = 0.69  -> tau_hi = min(0.72, 0.69) =
0.69
  Q_0.90(Eq)   = 0.81                         -> tau_eq = 0.81
Use §3.3 mapping with these bands.
```

QA checklist (L0–L2 for bands).

```
# L0: determinism
re-run calibration with same D_ref and split_seed -> same (tau_hi, tau_eq)

# L1: monotonicity w.r.t. alpha
decrease alpha_gt or alpha_eq -> non-decreasing tau_hi or tau_eq

# L2: holdout error bounds
err_gt <= alpha_gt + tiny_tol
err_eq <= alpha_eq + tiny_tol
```

SECTION 4. BRIDGES

What this section covers. Quick crosswalks showing how **SSMS** plugs into the **SSM** numeral and the unified environment **without reprinting theory**:

- **Structure rules:** elementwise ops, dot product, and matrix multiply built from **s_mul → s_sum** with streaming **U, W**.
- **Polynomials & functions:** composition via **s_pow** and **s_unary**.
- **Decision flows: s_gt/s_eq** scores with banded reporting, optional gate on alignment only.
- **Reproducibility:** one **manifest** across layers (same clamps, weights, and gate knobs).

4.1 Structural Composition (vectors, dot, matrix multiply)

Elementwise (any operator):
Apply the chosen SSMS operator to each component independently. Example for addition:
z_i = s_add(x_i, y_i) for all i

Dot product (s_mul → s_sum):
Given x_i=(m_i,a_i), y_i=(n_i,b_i)
t_i = s_mul(x_i, y_i)
dot(x,y) = s_sum({ t_i }) # maintain streaming U, W, m

Matrix multiply (cell-wise s_mul → s_sum):
For C = A x B,
C[i,j] = s_sum({ s_mul(A[i,k], B[k,j]) for k=1..K }) # streaming-safe

Collapse checks:
phi(z_i) = phi(x_i) op phi(y_i) (elementwise)
phi(dot(x,y)) = sum_i (m_i * n_i)
phi(C[i,j]) = sum_k (phi(A[i,k]) * phi(B[k,j]))

Guards and notes:

- Keep U, W accumulators per aggregate (dot or each C[i,j]) to preserve associativity.
- Zero-magnitude cells display as (0, +1) per zero-class policy.
- Optional gate applies after each inner op or once at the end (alignment-only).

4.2 Polynomials & Functions (composition via s_pow and s_unary)

Polynomials (definition):
Given x = (m, a) and coefficients c_k (use numerals (c_k, +1) unless alignment is intentional), define
p(x) = s_sum({ s_mul((c_k, +1), s_pow(x, k)) for k = 0..n })

Horner form (streaming-friendly):

```
y := (0, +1)
for k = n down to 0:
   y := s_add( s_mul(y, x), (c_k, +1) )
return y
```

Collapse check:
phi(p(x)) = sum_{k=0..n} c_k * m^k (classical polynomial)
phi(Horner(x)) = same result.

Domain and bounds:

- s_pow domain as specified (m > 0 for non-integer powers; any m for integer k).
- Alignment stays in (-1, +1) by existing operator bounds.
- Coefficients with alignment propagate via M2; use (c_k, +1) for purely classical coefficients.

Functions (general monotone f):
Use s_unary(f, x):
S_f(m) = m * f'(m) / f(m)
s_unary(f, x) = (f(m), tanh(S_f(m) * atanh(clamp_a(a, eps_a))))

Notes:

- s_unary recovers s_pow when f(m) = m^r; examples: sqrt, exp, log (with usual domain guards).
- Optional gate may be applied after evaluation to alignment only (a_env = g_t * a_op).

4.3 Decision Flows (scores -> bands -> gate)

Inputs. Numerals `x=(m_x,a_x)`, `y=(m_y,a_y)`; thresholds `tau_hi`, `tau_eq`; comparison knobs `beta_v >= 0`, `lambda_u >= 0`, `beta_u >= 0`; guards `V_eps > 0`, `eps_a > 0`; optional gate `g_t in [0,1]`.

Preferred evaluation order (alignment-aware).

```
1) dv  = ( m_x - m_y ) / max( |m_x| + |m_y| , V_eps )

2) u_x = atanh( clamp_a(a_x, eps_a) )
   u_y = atanh( clamp_a(a_y, eps_a) )

3) if gating enabled:
     du  = g_t * (u_x - u_y)
     d_u = g_t * |u_x - u_y|
   else:
     du  = (u_x - u_y)
     d_u = |u_x - u_y|

4) Scores:
   s_gt(x,y) = tanh( beta_v * dv + lambda_u * du )      # antisymmetric in
(x,y)
   raw_eq    = beta_v * abs(dv) + beta_u * d_u          # nonnegative
   s_eq(x,y) = 1 - 2 * tanh( raw_eq )                   # symmetric; in (-
1, +1]
```

Banded report (declare once; equality has precedence).

```
if s_eq >= tau_eq:           "equal"
else if s_gt >= +tau_hi:     "x > y"
else if s_gt <= -tau_hi:     "x < y"
else:                        "undecided"
```

Notes.

- **Collapse parity.** With `lambda_u = 0` and `beta_u = 0` (or `a_x = a_y`), scores reduce to classical magnitude comparisons via `dv`.
- **Gate placement.** Pre-score gating (step 3) attenuates alignment influence without touching magnitudes or `dv`. Use `g_t = 1` when no gating is desired.
- **Ranges & symmetry.**
- `s_gt in (-1, +1),  s_gt(y,x) = -s_gt(x,y)`
- `s_eq in (-1, +1],  s_eq(y,x) =  s_eq(x,y)`
- **Scale invariance.** For any `c > 0`, replacing `(m_x,m_y)` by `(c*m_x, c*m_y)` leaves `dv` (hence reports) unchanged.

Manifest (publish knobs used).

```
V_eps, eps_a
beta_v, lambda_u, beta_u
tau_hi, tau_eq
gate_used in {"on","off"} and gate knobs if "on" (lane_recipe, kappa, mu,
rho, Q0)
```

Minimal QA (L0–L1).

```
# L0 bounds
assert -1 < s_gt(x,y) < 1
assert -1 < s_eq(x,y) <= 1

# L0 antisymmetry/symmetry
assert s_gt(x,y) == -s_gt(y,x)
assert s_eq(x,y) ==  s_eq(y,x)

# L1 precedence
if s_eq >= tau_eq: report == "equal"  # regardless of s_gt

# L1 gating monotonicity (sampled)
for fixed x,y with u_gap = |u_x - u_y|:
  if g1 <= g2: then |s_gt| with g1 <= |s_gt| with g2  and  s_eq with g1 >= 
s_eq with g2
```

4.4 Reproducibility & Manifest (cross-layer)

One manifest for all layers.
This section lists configuration keys and data-provenance only; theoretical details are referenced elsewhere. Treat the manifest as the single source of truth.

Core keys (recap only).

```
gamma, eps_a, eps_w

division_policy      # "strict" | "meadow" | "soft"   (default "strict")
denom_soft_min       # > 0; required if and only if division_policy == 
"soft"
meadow_div           # legacy alias for "meadow"; canonical key is 
division_policy

gate_used            # "on" | "off"
lane_recipe          # string id; required iff gate_used == "on"
kappa, mu            # >= 0 ; gate knobs
rho                  # in [0,1] ; gate memory
Q0                   # in [0,1] ; initial memory state

zero_class_display   # bool ; show m==0 as (0,+1)
init_U, init_W       # streaming accumulators init
```

Comparison keys (declare once).

```
V_eps                # >0 ; floor in s_gt normalization
beta_v, lambda_u     # >=0 ; s_gt weights (magnitude vs alignment)
beta_u               # >=0 ; s_eq alignment weight
tau_hi, tau_eq       # in (0,1) ; decision bands (">/<") and equality
alpha_gt, alpha_eq   # in (0,1) ; band calibration risks
ref_size             # int ; |D_ref| used for calibration
split_seed           # int ; reproducible split for holdout
policy_gate          # "on" | "off" ; gate state used during calibration; 
must equal gate_used
```

Helper keys (optional).

```
V_unit              # legacy for s_round_k ; prefer V_eps in comparisons
tau_eq_mag          # legacy magnitude-only tie band ; prefer s_eq +
tau_eq
seed                # PRNG seed for demos/tests
run_id              # human-readable run tag
timestamp_utc       # ISO-8601 "YYYY-MM-DDTHH:MM:SSZ"
```

Data & provenance.

```
dataset_name, dataset_version
source_url                 # if external/public
preprocessing_notes        # scaling, normalization, filters, etc.
```

Minimal JSON skeleton (strict division, gate off, pre-calibrated bands).

```
{
  "gamma": 0.0,
  "eps_a": 1e-6,
  "eps_w": 1e-12,

  "division_policy": "strict",
  "denom_soft_min": null,
  "meadow_div": false,

  "gate_used": "off",
  "lane_recipe": null,
  "kappa": 0.0,
  "mu": 0.0,
  "rho": 0.0,
  "Q0": 0.0,

  "zero_class_display": true,
  "init_U": 0.0,
  "init_W": 0.0,

  "V_eps": 1e-12,
  "beta_v": 1.0,
  "lambda_u": 1.0,
  "beta_u": 1.0,

  "tau_hi": 0.70,
  "tau_eq": 0.80,
  "alpha_gt": 0.05,
  "alpha_eq": 0.10,
  "ref_size": 1000,
  "split_seed": 1729,
  "policy_gate": "off",

  "V_unit": null,
  "tau_eq_mag": null,

  "seed": 0,
  "run_id": "demo-001",
  "timestamp_utc": "YYYY-MM-DDTHH:MM:SSZ",

  "dataset_name": "N/A",
  "dataset_version": "N/A",
```

```
  "source_url": "N/A",
  "preprocessing_notes": "none"
}
```


Publishing requirement.
Ship the manifest with every example (and any CSV/JSON outputs). The manifest must reflect the exact knobs, policies, data versions, and calibration parameters used to produce reported numbers. Freeze it per release to ensure exact reproducibility.

SECTION 5. EXAMPLES

What this section covers. Three short, copy-ready, ASCII-only examples that demonstrate how SSMS operators work in practice while preserving **collapse parity** and **bounded alignment**.

Complementarity note: SSMS complements **Shunyaya Symbolic Mathematics (SSM)**. SSM defines the numeral **(m, a)** and the collapse map **phi**, and provides case studies based on real, publicly available datasets used under their respective licenses. For details and case studies, please refer to **Shunyaya Symbolic Mathematics_ver2.3**.

Independence & license: implementations are independent (no registry/keys/services); conformance = emit symbols and compute operators exactly as defined in the **Operators section** of this spec; optional blocks (privacy/gate/on-chain) are non-normative; spec is **CC BY 4.0** (creativecommons.org/licenses/by/4.0/); any third-party datasets retain their original licenses with attribution (no endorsement).

5.1 E1 - Alignment-Aware Averaging (s_sum)

Inputs (declare once): eps_a = 1e-6, eps_w = 1e-12, gamma >= 0.
Values: x_i = (m_i, a_i), i = 1..n. **Weights:** w_i = |m_i|^gamma.

Computation (batch form):
U = sum_i [w_i * atanh(clamp_a(a_i, eps_a))]
W = sum_i w_i
m_out = sum_i m_i
a_out = tanh(U / max(W, eps_w))

Streaming form (append new items):
U := U + sum_new w_i * atanh(clamp_a(a_i, eps_a))
W := W + sum_new w_i
m := m + sum_new m_i
a := tanh(U / max(W, eps_w))

Collapse check: phi(s_sum({x_i})) = sum_i m_i.
Bounds: |a_out| < 1 (clamp + tanh).
Note: gamma = 0 gives equal-weighted rapidity pooling; gamma > 0 emphasizes larger |m|.

Numeric micro-check (optional):
x1 = (3.0, +0.80), x2 = (1.0, +0.20), gamma = 0
U = 1.301344843 ; W = 2 ; m_out = 4.0 ; a_out = tanh(0.6506724215) = 0.572122462
Result: (4.0, 0.572122462) ; phi = 4.0

5.2 E2 - One Cell of 2x2 Matmul (s_mul -> s_sum)

Inputs:
A = [[A11, A12], [A21, A22]], B = [[B11, B12], [B21, B22]] with entries (m, a).
Goal: C[1,1] = A[1,:] dot B[:,1].

Steps:
t1 = s_mul(A11, B11)
t2 = s_mul(A12, B21)
C11 = s_sum({ t1, t2 }) # maintain U, W per cell (streaming-safe)

Collapse check:
phi(C11) = phi(A11)*phi(B11) + phi(A12)*phi(B21) # classical dot product

Bounds and notes:
• Each **s_mul** uses rapidity addition on alignment (M2), so **|a| < 1** persists.
• **s_sum** uses U, W accumulators; order of addition does not change the result (associativity preserved in streaming).
• **Optional gate** may be applied after C11 is computed: a_env = g_t * a_C11 ; m unchanged.

Numeric micro-check (optional):
A11 = (2.0, +0.60), B11 = (5.0, +0.50) ; A12 = (1.0, +0.30), B21 = (4.0, -0.20)
t1 = (10.0, 0.846153846) ; t2 = (4.0, 0.106382979)
U = 1.349240375 ; W = 2 ; m_out = 14.0 ; a_out = tanh(0.674620188) = 0.588010822
Result: C11 = (14.0, 0.588010822) ; phi = 14

5.3 E3 - s_unary(sqrt) vs s_pow(r = 0.5)

Inputs: x = (m, a) with m > 0 ; eps_a = 1e-6.

Using s_pow:
s_pow(x, 0.5) = (m^0.5 , tanh(0.5 * atanh(clamp_a(a, eps_a))))

Using s_unary with f(m) = sqrt(m):
S_f(m) = m * f'(m) / f(m) = 1/2
s_unary(f, x) = (sqrt(m) , tanh(0.5 * atanh(clamp_a(a, eps_a))))

Conclusion: s_unary(sqrt) and **s_pow(0.5)** produce identical alignment propagation.
Collapse check: phi(...) = sqrt(m) in both cases.
Note: The same equivalence holds for f(m) = m^r with S_f(m) = r (domain guards apply).

Numeric micro-check (optional):
x = (9.0, +0.60) -> both paths return (3.0, 0.333333333) ; phi = 3.0.

SECTION 6. APPENDIX — MANIFEST & QA

What this section covers. A one-page **manifest crib** (keys grouped for quick publishing) and a concise **QA checklist** to verify collapse parity, bounds, and streaming safety without reprinting theory.

6.1 Manifest Crib (publish once per study)

One manifest for all layers.
This section lists configuration keys and data-provenance only; theoretical details are referenced elsewhere. Publish the exact knobs and provenance used for SSM numerals, SSMS operators, comparisons, and (if used) the environment gate.

Core keys (recap only).

```
gamma                 # >= 0 ; pooling weights: w_i = |m_i|^gamma
eps_a                 # alignment clamp margin (recommend 1e-6)
eps_w                 # denominator guard for pooling (recommend 1e-12)

division_policy       # "strict" | "meadow" | "soft"  (default "strict")
denom_soft_min        # > 0 ; required if and only if division_policy ==
"soft"

# compatibility alias (accept but prefer division_policy)
meadow_div            # bool ; synonym for division_policy == "meadow"
```

Gate (if used).

```
gate_used             # "on" | "off" ; alignment-only gate after each op
lane_recipe           # short id naming how Z_t and A_t are computed (e.g.,
"ZE-lane-v1")
kappa, mu             # >= 0 ; gate knobs
rho                   # in [0,1] ; gate memory
Q0                    # in [0,1] ; initial memory state
```

Comparisons (if used).

```
V_eps                 # > 0 ; floor in s_gt normalization
beta_v, lambda_u      # >= 0 ; s_gt weights (magnitude vs alignment)
beta_u                # >= 0 ; s_eq alignment weight
tau_hi, tau_eq        # in (0,1) ; reporting bands
alpha_gt, alpha_eq    # in (0,1) ; band calibration risks (quantile rule)
ref_size              # int ; |D_ref| used for calibration
split_seed            # int ; reproducible train/holdout split
policy_gate           # "on" | "off" ; gate state used during calibration;
must equal gate_used
```

Display and streaming.

```
zero_class_display    # bool (default true; show m == 0 as (0, +1))
init_U, init_W        # streaming accumulator init for sums (defaults 0.0)
```

Helpers (optional).

```
V_unit                # quantization scale for s_round_k (legacy)
tau_eq_mag            # magnitude-only tie band for min/max (legacy)
```

Repro metadata (recommended).

```
seed, run_id, timestamp_utc
dataset_name, dataset_version, source_url, preprocessing_notes   # "N/A" if
none
```

Minimal JSON skeleton (strict division, gate off, pre-calibrated bands).


```
{
  "gamma": 0.0,
  "eps_a": 1e-6,
  "eps_w": 1e-12,

  "division_policy": "strict",
  "denom_soft_min": null,
  "meadow_div": false,

  "gate_used": "off",
  "lane_recipe": null,
  "kappa": 0.0,
  "mu": 0.0,
  "rho": 0.0,
  "Q0": 0.0,

  "zero_class_display": true,
  "init_U": 0.0,
  "init_W": 0.0,

  "V_eps": 1e-12,
  "beta_v": 1.0,
  "lambda_u": 1.0,
  "beta_u": 1.0,
  "tau_hi": 0.70,
  "tau_eq": 0.80,
  "alpha_gt": 0.05,
  "alpha_eq": 0.10,
  "ref_size": 1000,
  "split_seed": 1729,
  "policy_gate": "off",

  "V_unit": null,
  "tau_eq_mag": null,

  "seed": 0,
  "run_id": "demo-001",
  "timestamp_utc": "YYYY-MM-DDTHH:MM:SSZ",

  "dataset_name": "N/A",
  "dataset_version": "N/A",
```

```
  "source_url": "N/A",
  "preprocessing_notes": "none"
}
```

Publishing requirement.
Ship this manifest next to every example/CSV/JSON. It is the single source of truth for knobs, policies, datasets, and calibration. Freeze per release.

6.2 QA Checklist (run once per study/release)

Q1. Collapse parity (all operators).
For random `x,y,...` (and tensors), verify:

```
phi( s_op(x, y, ...) ) == classical_op( phi(x), phi(y), ... )
```

For division, match the selected policy:

```
phi( s_div(x, y) ) == classical_div_policy( phi(x), phi(y) )
```

Q2. Bounds (alignment).
After every op (and after optional gate), check:

```
|a_out| <= 1 - eps_a    # due to clamp_a + tanh
```

Q3. Addition associativity (streaming vs batch).
Compute `S_batch = s_sum({x_i})`.
Compute `S_stream` by appending `x_i` with maintained `(U, W, m)`:

```
assert phi(S_batch) == phi(S_stream)
assert |a_batch - a_stream| <= 1e-12
```

Q4. M2 properties (mul/div).
Check `s_mul` commutativity and associativity on alignment (within tolerance):

```
tanh( u(a1)+u(a2) ) == tanh( u(a2)+u(a1) )
tanh( (u(a1)+u(a2))+u(a3) ) == tanh( u(a1)+(u(a2)+u(a3)) )
```

Division definedness per policy:

```
strict:  require m_y != 0
meadow:  totalized; inv_m(0)=0
soft:    use denom = sign_star(m_y) * max(|m_y|, denom_soft_min)
```

Q5. Identities & inverses.

```
s_add(x, id_add) == x
s_mul(x, id_mul) == x
```

Right-inverse on magnitudes:

```
strict:  if m_x != 0,  s_mul(x, s_inv_mul(x)) -> id_mul on magnitudes
meadow:  if m_x != 0,  same; if m_x == 0, magnitude -> 0  (display (0,+1)
if enabled)
soft:    if |m_x| >= denom_soft_min, -> id_mul on magnitudes
         else magnitude -> m_x / (sign_star(m_x)*denom_soft_min)  #
bounded, not 1
```

Q6. Zero-class display.
Whenever `m_out == 0`, final **display** is `(0, +1)` (do not alter internal alignment).

Q7. Domain guards (pow/unary).

```
s_pow(x, r):
  - if r is non-integer, require m_x > 0 (classical domain)
  - if r < 0: require m_x != 0 under strict; meadow/soft allowed by policy

s_unary(f, x):
  - ensure f and f' defined on phi(x)
  - avoid zeros in denominators; if unavoidable, clip or publish a guard
for that op
```

Q8. Gate invariants (if used).

```
g_t in [0,1]
phi( (m_op, clamp_a(g_t*a_op, eps_a)) ) == m_op
|clamp_a(g_t*a_op, eps_a)| <= |a_op|            # monotone calming
if g1 <= g2:  |a_env(g1)| <= |a_env(g2)|        # sampled monotonicity
```

Q9. Comparison sanity (s_gt, s_eq).

```
lambda_u = beta_u = 0   ->   sign( s_gt(x,y) ) = sign( m_x - m_y )
s_eq symmetric:              s_eq(x,y) == s_eq(y,x)
s_gt antisymmetric:          s_gt(x,y) == -s_gt(y,x)
Band edges honored:          transitions at tau_hi, tau_eq
Scale invariance:            report(c*m_x, c*m_y) == report(m_x, m_y) for
c>0
```

Q10. Helpers consistency.

```
s_min/s_max collapse to classical min/max
tie case (m_x == m_y): pooled alignment via s_sum with declared weights
s_round_k collapses to round(m, k); |a| is nonincreasing after quantization
```

Q11. Determinism & manifest.

```
With fixed manifest (and seed, if used), repeated runs are identical
All required keys present; unknown keys ignored or logged without effect
Gate state during evaluation matches policy_gate used for band calibration
```

Q12. Numerical safety.

```
No NaN/Inf from atanh/tanh due to clamp_a
Denominators guarded:
  - pooling:      max(W, eps_w)
```

```
  - s_gt dv:      max(|m_x|+|m_y|, V_eps)
  - soft division: sign_star(m_y)*max(|m_y|, denom_soft_min)
```

Tiny harness sketch (addition test).

```
# batch
U = sum_i w_i * atanh( clamp_a(a_i, eps_a) )
W = sum_i w_i
m = sum_i m_i
a = tanh( U / max(W, eps_w) )
S_batch = (m, a)

# streaming
U=W=m=0
for each i:
  U += w_i * atanh( clamp_a(a_i, eps_a) )
  W += w_i
  m += m_i
a_stream = tanh( U / max(W, eps_w) )
S_stream = (m, a_stream)

assert phi(S_batch) == phi(S_stream)
assert abs(a - a_stream) <= 1e-12
```

Tiny harness sketch (division policy smoke tests).

```
# strict
assert raises_domain_error( s_div( (m1,a1), (0,a2) ) )

# meadow
out = s_div( (m1,a1), (0,a2) )
assert phi(out) == 0
assert -1 < a_out < 1

# soft
denom_soft_min = 1e-6
out = s_div( (m1,a1), (m2≈0,a2) )
assert is_finite( phi(out) )
```

Pass criterion.
All assertions hold across randomized and edge-case suites (zeros, large/small magnitudes, alignments near +/-1, extreme policy settings). Record manifest, seed, and data references with the test artifacts.

Conformance note.
Passing the full suite qualifies the implementation for the **SSMS-1.8-Conformant** badge (see Resources).

SECTION 7. FUTURE WORK

What this section covers. Near-term extensions that build on the SSMS operator canon without reprinting theory. All items remain observation-only, collapse-safe, and manifest-first.

7.1 Five-Element Transition Layer (deferred)

A bounded, per-step lens that modulates alignment within each integer step (… -2->-1->0->1->2 …). Valuable idea, but deferred until a full QA harness and reproducibility kit are ready. Not included in v1.8.

7.2 Domain Adapters (family roadmap)

Because **Shunyaya Symbolic Mathematics (SSM)** with **Shunyaya Symbolic Mathematical Symbols (SSMS)** complements the core foundations of mathematics, the canon applies across domains and aspects of life. Just as with SSM-Chem, future releases can instantiate the same operators and manifest in other domains—such as **SSM-Physics, SSM-Geo, SSM-Time, SSM-Cyber, SSM-AI, SSM-Health, SSM-Fin, SSM-Climate, SSM-Robotics, SSM-Controls**—with domain-specific contrast recipes and brief notes, while avoiding theory reprints.

7.3 Mini-Spec Template (2 pages per domain)

1. **Core objects:** what becomes x = (m, a).
2. **Contrasts -> alignment:** e = (score_fit - score_violate) / U ; a = tanh(c * e).
3. **Stability index:** bounded index in (-1, +1).
4. **Gate (optional):** a_env = g_t * a (magnitudes unchanged).
5. **Drop-ins:** r = (1 + a)/2 scaling; banded decisions via s_gt/s_eq.
6. **One micro-example:** synthetic, collapse-safe, ASCII; manifest included.

7.4 Band Calibration Guide (short protocol)

Pick **tau_hi** and **tau_eq** from validation sweeps; publish chosen bands and include a tiny sanity table.

7.5 Example Pack (synthetic, reproducible)

Small CSVs with reference outputs for **s_sum**, **s_mul -> s_sum**, **s_gt/s_eq** (with and without gating). Include manifest snapshots for exact replay.

7.6 Governance & Licenses

One manifest per study. If future examples use external datasets, use only publicly available data under their licenses with explicit attribution; no endorsement implied.

7.7 Versioning & Change Policy

- **v0.x:** operator canon stable; add adapters, examples, and docs only.
- **v1.0 target:** finalize Tier 1 adapters, band calibration guide, QA harness; consider promoting the deferred transition layer once reproducibility is demonstrated.

---

8. VERSION HISTORY

V1.8 — brief highlights

• **Benefits & Complementarity:** tightened wording; merged zero/identity determinism into classical compatibility (phi(0, a) = 0; phi(1, a) = 1); all formulas in plain ASCII.
• **Observation-only reminders:** kept scope explicit in Abstract; added non-normative Appendix D footnote for stricter neutrality at dv ~ 0 (|dv| <= tau_small -> "undecided").
• **Appendix D (demo):** documentation clarifications only (equality-first precedence, manifest note); **HTML/JS left unchanged**.
• **Combine & clamps:** emphasized bounded combine policies (**M1: a' = a1 * a2; M2: a' = tanh(atanh(a1) + atanh(a2))**) and clamp guard (**|a| <= 1 - eps_a**).
• **Division note:** s_div safety policy remains in operator section; at-a-glance list trimmed for readability.
• **Manifest & reproducibility:** minimal, audit-ready knobs unchanged; wording refined for consistency.
• **License & formatting:** CC BY 4.0 retained; phrasing and headings polished for GitHub readme rendering.

Appendix A — Unified Manifest Skeleton (single source of truth)

Purpose. One minimal manifest that drives SSM/SSMS operators, gating, banded decisions, and performance modes. Exact math is default; approximations and gates are explicit and alignment-only.

Recommended defaults (may be overridden per study).
gamma = 0
eps_a = 1e-6
eps_w = 1e-12
division_policy = "strict"
zero_class_display = true

JSON skeleton (copy/paste).


```
{
  "version": "ssms-1.8",
  "study_id": "your-study-id",
  "canon": {
    "gamma": 0,
    "eps_a": 1e-6,
    "eps_w": 1e-12,
    "zero_class_display": true,
    "division_policy": "strict",
    "meadow_div": false,
    "denom_soft_min": null
  },
  "gate_used": false,
  "gate": {
    "g_mode": "constant",
    "g_min": 0.05,
    "mu": 0.0,
    "kappa": 1.0
  },
  "banding": {
    "V_unit": 1.0,
    "tau_eq_mag": 0.0,
    "tau_hi": 3.0,
    "tau_align": null
  },
  "approx": {
    "approx_mode": "exact",
    "a_poly_max": 0.8,
    "u_rational_max": 3.0,
    "u_sat": 4.0,
    "target_abs_err": 1e-3
  },
  "error_channel": {
    "error_channel_used": false,
    "lambda_e": 0.0,
    "g_min": 0.05,
    "error_metric": null,
    "error_unit": null
  },
  "linalg": {
    "A_beta_target": null,
    "U_ceiling": null,
    "gate_policy": "global"
  },
  "init_U": 0.0,
  "init_W": 0.0
}
```


Field glossary (ultra-short).
• gamma: weight exponent in w = |m|^gamma for sums.
• eps_a: alignment clamp; enforce |a| <= 1 - eps_a.
• eps_w: denominator guard for W in U/W pooling.
• zero_class_display: if m == 0, display (0, +1).
• division_policy: "strict" or "meadow" or "soft" (see 2.13).
• gate_used plus gate.g_mode/g_min/mu/kappa: unified gate (alignment-only).
• V_unit, tau_eq_mag, tau_hi, tau_align: banded comparison (2.8).

• approx_* and target_abs_err: big-data performance (2.11).
• error_channel_*: s_error configuration (2.15).
• A_beta_target, U_ceiling, gate_policy: linear-algebra stability (2.12).
• init_U, init_W: streaming accumulator initialization for sums.

Invariant reminders.
• Collapse parity: phi depends only on magnitudes; gates and approximations never change phi.
• Clamp discipline: always pre-clamp alignment before atanh; clamp final a.
• Determinism: same inputs + same manifest = identical outputs.

Appendix B - QA & Release Checklist (Go/No-Go)

Purpose. A minimal, deterministic checklist to certify SSMS builds. Exact math is default; approximations and gates are explicit and alignment-only. Use this as your release gate.

B.1 Core invariants (must pass)
• Collapse parity: for every tested op F, phi(F(inputs)) == classical_result_on_magnitudes.
• Bounds: for every output, |a_out| <= 1 - eps_a after clamp.
• Determinism: same inputs + same manifest -> identical outputs (bit-stable within platform precision).

B.2 Manifest completeness (single source of truth)
• Required keys present and typed: gamma, eps_a, eps_w, division_policy, zero_class_display.
• If gate_used != null: g_mode, g_min, (mu or kappa if used).
• If banding used: V_unit, tau_eq_mag, tau_hi (and optional tau_align).
• If approx_mode != "exact": a_poly_max, u_rational_max, u_sat, target_abs_err.
• If error_channel_used: lambda_e, g_min, error_metric, error_unit.
• If linalg tracking: A_beta_target, U_ceiling, gate_policy.

B.3 Operator tests (unit)
• s_sum: random batches (sizes 1..N). Check phi == sum(m_i). Check W guard: W >= eps_w.
• s_add/s_sub: pairwise against s_sum; s_sub(x,y) == s_add(x, s_neg(y)).
• s_mul/s_div: phi == m1*m2 and m1/m2 (per division_policy). Alignment via u add/sub.
• s_pow: integers and a few non-integers (domain guards). phi == m^r.
• s_unary: f in {sqrt, exp, log (m>1)}. Check S_f use and phi == f(m).
• helpers: min/max/abs/round behave per definitions and preserve bounds.

B.4 Gate tests (alignment-only)
• Per-op vs end-of-block with constant g_t: identical a_out (within clamp tolerance).
• Varying g_t over time: product rule holds, i.e., sequential gates multiply.
• Verify phi unchanged by gating.

B.5 Decision layer tests (banded comparisons)

• Construct delta_m grids around 0 with chosen V_unit, tau_eq_mag, tau_hi.
• Expected labels:
|delta_m| <= tau_eq_mag -> "equal";
|delta_m|/V_unit >= tau_hi AND (|a_dec| >= tau_align if used) -> "x>y" or "x<y";
else -> "undecided".
• Verify pooled alignment a_dec is clamped and gate-attenuated only.

B.6 Tensor and linear algebra tests

• s_einsum: matmul "ik,kj->ij" against classical phi; random shapes 2x2, 3x3.
• Streaming equivalence: incremental (U,W,m) matches batch within tolerance on a.
• A_beta bound: compute empirical ||u_y||_inf and verify
||u_y||_inf <= A_beta + ||u_x||_inf (no gate),
||u_y_env||_inf <= |g_t|*(A_beta + ||u_x||_inf) (with gate).

B.7 Performance mode tests (if approx enabled)

• Grid on a in (-1+eps_a, 1-eps_a):
|atanh_approx(a) - atanh_exact(a)| <= target_abs_err inside declared domain; fallback exact outside.
• Grid on u in [-u_rational_max, +u_rational_max]:
|tanh_approx(u) - tanh_exact(u)| <= target_abs_err inside domain; fallback exact outside.
• End-to-end parity: s_sum, s_mul, s_pow, s_einsum produce identical phi and a within target_abs_err.

B.8 Determinism and reproducibility pack

• Include: manifest.json, inputs.csv (or .txt), outputs.csv, and a SHA256 of each file.
• Record platform info: language/runtime, BLAS (if any), float type.
• Provide a seed for any randomized tests; disallow non-deterministic threads.

B.9 Minimal harness (pseudo-spec)

• collapse_check(F, inputs): assert phi(F(inputs)) == classical(inputs.m).
• bounds_check(x): assert abs(a_out) <= 1 - eps_a + 1e-12.
• determinism_check(): run twice; byte-compare outputs (or numeric within 1e-12 for exact mode).
• performance_check(): if approx_mode != "exact", run B.7 grids.

B.10 Ethics and labeling

• Observation-only: no operational control claims.
• Disclose division_policy, gate policy, and approx_mode in the report header.
• If zero_class_display == true, state that (0,+1) is a display convention.

B.11 Go/No-Go rule (pass criteria)

Release = GO if and only if all of the following are true:

1. All B.3-B.7 tests pass.
2. Determinism (B.8) holds.
3. Report pack (B.8) is complete and checksummed.
4. Ethics/labels (B.10) present and correct.

Tolerance defaults
• Exact mode: numeric tolerance 1e-12 on alignment computations (post-tanh) and exact parity on phi.
• Approx mode: alignment abs error <= target_abs_err inside declared domains; exact parity on phi.

One-line reminder
• "If phi changes, it is a bug; if only a changes, it must be by explicit gate or declared approximation."

Appendix C — Glossary & Quickref (minimal, copy-ready)

Collapse map (phi).
Returns the classical magnitude and ignores alignment:

```
phi( (m, a) ) = m
```

All SSMS operators are designed so that collapsing the result reproduces the classical outcome wherever that classical operation is defined.

Symbolic numeral.
A pair carrying value and bounded stability signal:

```
x = (m, a)   with  m in R,  a in (-1, +1)
```

Clamp (alignment).
Bounds alignment strictly inside (-1, +1):

```
clamp_a(a, eps_a) = sign(a) * min( abs(a), 1 - eps_a )
eps_a = 1e-6   # default
```

Rapidity (linearized stability).

```
u(a) = atanh( clamp_a(a, eps_a) )
a(u) = tanh(u)
```

Zero-class (display rule).

```
if m == 0: display (0, +1)
phi((0, +1)) = 0
```

Identity elements.

```
id_add = (0, +1)
id_mul = (1, +1)
```

Gate (alignment-only attenuator; optional).
Applies after an operator; never changes magnitudes:

```
a_env = clamp_a( g_t * a_op , eps_a )
phi( (m_op, a_env) ) = m_op
```

Division policy (magnitude path).
Choose exactly one; alignment formulas are unchanged.

```
division_policy = "strict"  # default: require m2 != 0
division_policy = "meadow"  # totalize: inv_m(0) = 0
division_policy = "soft"    # guard small denominators
# soft helper:
denom = sign_star(m2) * max( |m2| , denom_soft_min )   # sign_star(0) := +1
```

Core operators (signatures & collapse).

• **Sum / pooling (streaming-safe).**

```
# weights w_i are nonnegative; common: w_i = 1  or  w_i = |m_i|^gamma
(gamma >= 0)
U = sum_i  w_i * u(a_i)
W = sum_i  w_i
m_out = sum_i m_i
a_out = tanh( U / max(W, eps_w) )
# collapse:
phi( s_sum({(m_i,a_i)}, {w_i}) ) = sum_i m_i
eps_w = 1e-12   # default
```

• **Multiply (M2 combine).**

```
s_mul( (m1,a1), (m2,a2) ) = ( m1*m2 , tanh( u(a1) + u(a2) ) )
# collapse:
phi( s_mul(x1,x2) ) = m1*m2
```

• **Reciprocal and division.**

```
s_inv_mul( (m,a) )         = ( inv_m(m) , tanh( -u(a) ) )
s_div( (m1,a1), (m2,a2) ) = ( m1 * inv_m(m2) , tanh( u(a1) - u(a2) ) )

# inv_m(m) by policy:
#   strict:  inv_m(m) = 1/m      (domain: m != 0)
#   meadow:  inv_m(m) = 0 if m == 0 else 1/m
#   soft:    inv_m(m) = 1 / ( sign_star(m) * max(|m|, denom_soft_min) )
```

• **Power (when classically defined).**

```
s_pow( (m,a), r ) = ( m^r , tanh( r * u(a) ) )
# collapse:
phi( s_pow(x, r) ) = (phi(x))^r
```

• **Min / Max (selection).**

```
s_min(x,y) = ( min(m_x, m_y) , a_selected )
s_max(x,y) = ( max(m_x, m_y) , a_selected )
```

```
# a_selected: alignment from the arg attaining the selected magnitude; ties
may pool by s_sum.
```

Comparisons (scores in (-1, +1)).

• Greater-than score (s_gt, antisymmetric).

```
V_eps > 0
dv = ( m_x - m_y ) / max( |m_x| + |m_y| , V_eps )
du = u(a_x) - u(a_y)
s_gt(x,y) = tanh( beta_v * dv + lambda_u * du )   # beta_v >= 0, lambda_u
>= 0
```

• Equality score (s_eq, symmetric).

```
raw_eq    = beta_v * abs(dv) + beta_u * abs(du)   # beta_u >= 0
s_eq(x,y) = 1 - 2 * tanh( raw_eq )
```

Banded reporting (global policy).

```
# thresholds in (0,1)
tau_hi  # for s_gt (">" / "<")
tau_eq  # for s_eq ("equal")

# mapping (precedence: equality first)
if s_eq(x,y) >= tau_eq:         "equal"
else if s_gt(x,y) >= +tau_hi:   "x > y"
else if s_gt(x,y) <= -tau_hi:   "x < y"
else:                           "undecided"
```

Band calibration (quantile rule; tiny protocol).

```
Inputs: D_ref = {(x_i,y_i,label_i)}, label in {">","<","="}
Compute:
  Gpos = { s_gt(x,y)   | label == ">" }
  Gneg = { -s_gt(x,y)  | label == "<" }
  Eq   = { s_eq(x,y)   | label == "=" }
Choose:
  alpha_gt in (0,1), alpha_eq in (0,1)
Set:
  tau_hi = min( Q_{1-alpha_gt}(Gpos), Q_{1-alpha_gt}(Gneg) )
  tau_eq =      Q_{1-alpha_eq}(Eq)
```

Streaming parity (reducers).
For reducers like `s_sum`, batch and append orders yield the same collapsed magnitude:

```
phi( s_sum(batch) ) == sum( phi(batch) )
phi( fold_stream(s_sum, stream) ) == sum( phi(stream) )
```

Unified manifest (keys to publish).

```
# guards
eps_a, eps_w, V_eps

# combine & compare
beta_v, lambda_u, beta_u
```

```
# division
division_policy in {"strict","meadow","soft"}
meadow_div (bool; synonym for "meadow")
denom_soft_min  # required if "soft"

# gate
gate_used in {"on","off"}   # fixed choice for release

# bands
tau_hi, tau_eq
alpha_gt, alpha_eq
ref_size, split_seed
policy_gate in {"on","off"}  # gate state used during calibration
```

Minimal QA (release checklist).

```
L0: collapse parity
  assert phi( s_op(args) ) == classical_op( phi(args) )
L0: bounds
  assert |a_out| <= 1 - eps_a
L1: streaming parity (s_sum)
  batch vs append: equal collapsed magnitudes
L1: division totality (if meadow) / floor (if soft)
  no NaN/Inf; floors respected
L2: bands holdout
  err_gt <= alpha_gt + tol  ;  err_eq <= alpha_eq + tol
```

Resources (external, zero-bloat).
- `ssms-ref library (reference implementation)`
- `ssms_manifest.schema.json + ssms-validate CLI (manifest JSON Schema and validator)`
- `SSMS-1.8-Conformant test pack (conformance badge)`
- `Band calibration kit (quantile rule)`
- `Interactive demo (Appendix D, single file)`

`Note: All items are non-normative; the spec text remains authoritative.`

Appendix D — Interactive Demo (single-file, non-normative)

Purpose. Explore alignment-aware comparisons with bands and an optional gate. Illustrative only; the spec text is normative. **Soft reminder (observation-only):** this demo is for analysis and decision support; it is **not** a control or safety system.

Footnote (non-normative): For stricter neutrality when dv ~ 0, apply the optional usage rule `|dv| <= tau_small -> report "undecided"`.

How to run. Save the code below as `ssms_demo.html` and open it locally with any modern browser.

What it shows. Inputs: `x=(m_x,a_x)`, `y=(m_y,a_y)`; knobs `beta_v, lambda_u, beta_u`; guards `eps_a, V_eps`; bands `tau_hi, tau_eq`; optional gate `g_t`. Outputs: `dv, du, s_gt, s_eq`, and the banded report.

Non-normative clarifications (no code changes):

- **Equality-first precedence.** The banded mapping treats equality first (`if s_eq >= tau_eq -> "equal"`). When `dv ≈ 0`, **alignment may tip** the greater-than score (`s_gt`). Teams wanting stricter neutrality can adopt a **usage rule** (outside the code): *force "undecided" whenever* `|dv| <= tau_small`, with a domain-chosen `tau_small` (e.g., `0.01` to `0.05`).
- **Reproducibility note.** The Manifest snippet mirrors the current knobs/inputs. If you need audit timestamps, record a **separate timestamp** alongside the copied JSON (the demo intentionally keeps the snippet minimal).
- **Accessibility note.** Sliders and number boxes are keyboard-friendly. For formal deployments, consider ARIA labels and slightly larger mono text; this demo keeps UI minimal by design.

```
<!doctype html>
<html lang="en">
<meta charset="utf-8">
<title>SSMS Interactive Demo (single file, auto-compute)</title>
<meta name="viewport" content="width=device-width,initial-scale=1">
<style>
  *, *::before, *::after { box-sizing: border-box; }
  html, body { width: 100%; max-width: 100%; overflow-x: hidden; }

  :root { --fg:#0f172a; --mut:#475569; --bg:#f8fafc; --card:#ffffff; --
brd:#e2e8f0; --hi:#e0f2fe; }
  body { margin:0; font-family:system-ui,-apple-system,Segoe
UI,Roboto,Ubuntu,Cantarell,Inter,Arial,sans-serif; color:var(--fg);
background:var(--bg); }
  .wrap { max-width:1000px; margin:auto; padding:24px; }
  h1 { font-size:22px; margin:0 0 8px; }
  p.sub { margin:0 0 16px; color:var(--mut); font-size:13px; }
  .grid { display:grid; gap:16px; }
  @media(min-width:900px){ .g2{ grid-template-columns:1fr 1fr; } }
  .card { background:var(--card); border:1px solid var(--brd); border-
radius:14px; padding:16px; }
  .title { font-weight:600; margin-bottom:8px; }

  label.row { display:grid; grid-template-columns:150px minmax(0, 1fr);
gap:10px; align-items:center; margin:8px 0; font-size:14px; }
  input[type="range"]{ width:100%; }
  input[type="number"]{ width:clamp(90px, 18vw, 120px); padding:6px;
border:1px solid var(--brd); border-radius:8px; font-size:13px; }
  .nums { display:grid; grid-template-columns:minmax(0,1fr) auto; gap:8px;
font-size:14px; }
  .mono { font-family:ui-monospace,SFMono-
Regular,Menlo,Monaco,Consolas,monospace; }

  /* Fixed action bar */
  .fixed-bar {
    position: fixed; top: 0; left: 0; right: 0; z-index: 1000;
    background: var(--bg);
    border-bottom: 1px solid var(--brd);
    padding: 10px 0;
```

```
    backdrop-filter: saturate(120%) blur(4px);
  }
  .bar-inner { max-width:1000px; margin:auto; padding: 0 24px; }
  .bar-spacer { height: 64px; } /* set on load to actual height */

  .actions { display:flex; gap:10px; flex-wrap:wrap; align-items:center; }
  .status { font-size:12px; color:var(--mut); }

  /* Buttons with pressed feedback */
  .btn { display:inline-flex; align-items:center; gap:6px; border:1px solid
var(--brd); background:#fff; padding:8px 12px; border-radius:10px;
cursor:pointer; font-size:14px; user-select:none; transition: transform
60ms ease, box-shadow 60ms ease, background 120ms ease; }
  .btn:active, .btn.is-pressed { transform: translateY(1px); box-shadow:
inset 0 2px 6px rgba(0,0,0,.15); }
  .btn:focus-visible { outline:2px solid #94a3b8; outline-offset:2px; }

  /* Mini results chip */
  .chip { display:inline-flex; align-items:center; gap:8px; border:1px
solid var(--brd); background:#fff; padding:6px 10px; border-radius:999px;
font-size:12px; }
  .chip .dot { width:8px; height:8px; border-radius:999px;
background:#0ea5e9; }
  @keyframes chipPop { 0%{ transform:scale(.98); } 50%{
transform:scale(1.02); } 100%{ transform:scale(1); } }
  .chip-pop { animation: chipPop 180ms ease-out 1; }

  /* Results highlight pulse */
  @keyframes pulse {
    0% { box-shadow: 0 0 0 0 rgba(14,165,233,.45); background:var(--hi); }
    50% { box-shadow: 0 0 0 8px rgba(14,165,233,.0); background:#f1f5f9; }
    100% { box-shadow: none; background:var(--card); }
  }
  .pulse-once { animation: pulse 900ms ease-out 1; }
</style>
<body>
  <!-- Fixed header -->
  <div class="fixed-bar" id="fixed_bar">
    <div class="bar-inner">
      <div class="actions">
        <button class="btn" type="button" id="reset_btn"
onclick="resetDefaults()">Reset defaults</button>
        <span class="status" id="status" role="status" aria-
live="polite">Auto compute: on | Last updated: —</span>
        <span class="chip" id="mini_chip" aria-live="polite">
          <span class="dot"></span>
          <span id="mini_out">s_gt=—, s_eq=—, report=—</span>
        </span>
      </div>
    </div>
  </div>
  <!-- Spacer so content isn't hidden under the fixed bar -->
  <div class="bar-spacer" id="bar_spacer"></div>

  <div class="wrap">
    <h1>SSMS Interactive Demo (single file)</h1>
    <p class="sub">Explore alignment-aware comparisons with bands and an
optional gate. Formulas shown in plain ASCII. Illustrative; the spec is
normative.</p>

    <div class="grid g2" style="margin-top:12px;">
```

```
      <div class="card">
        <div class="title">Inputs: numerals x=(m_x,a_x), y=(m_y,a_y)</div>
        <label class="row"><span>m_x</span>
          <div>
            <input id="mx_range" type="range" min="-100" max="100"
step="0.1" value="5">
            <div class="nums"><input id="mx_num" type="number" step="0.1"
value="5"><span class="rowhint">real (R)</span></div>
          </div>
        </label>
        <label class="row"><span>a_x in (-1,1)</span>
          <div>
            <input id="ax_range" type="range" min="-0.999" max="0.999"
step="0.001" value="0.4">
            <div class="nums"><input id="ax_num" type="number" step="0.001"
value="0.4"><span class="rowhint">bounded</span></div>
          </div>
        </label>
        <label class="row"><span>m_y</span>
          <div>
            <input id="my_range" type="range" min="-100" max="100"
step="0.1" value="3">
            <div class="nums"><input id="my_num" type="number" step="0.1"
value="3"><span></span></div>
          </div>
        </label>
        <label class="row"><span>a_y in (-1,1)</span>
          <div>
            <input id="ay_range" type="range" min="-0.999" max="0.999"
step="0.001" value="0.1">
            <div class="nums"><input id="ay_num" type="number" step="0.001"
value="0.1"><span></span></div>
          </div>
        </label>
        <pre class="mono">Clamp:    clamp_a(a, eps_a) = sign(a) * min( |a|,
1 - eps_a )
Rapidity: u(a) = atanh( clamp_a(a, eps_a) )</pre>
      </div>

      <div class="card">
        <div class="title">Knobs & bands</div>
        <label class="row"><span>beta_v (>=0)</span>
          <div>
            <input id="beta_v_range" type="range" min="0" max="4"
step="0.1" value="1.0">
            <div class="nums"><input id="beta_v_num" type="number"
step="0.1" value="1.0"><span class="rowhint">magnitude weight</span></div>
          </div>
        </label>
        <label class="row"><span>lambda_u (>=0)</span>
          <div>
            <input id="lambda_u_range" type="range" min="0" max="4"
step="0.1" value="1.0">
            <div class="nums"><input id="lambda_u_num" type="number"
step="0.1" value="1.0"><span class="rowhint">alignment weight</span></div>
          </div>
        </label>
        <label class="row"><span>beta_u (>=0)</span>
          <div>
            <input id="beta_u_range" type="range" min="0" max="4"
step="0.1" value="1.0">
```

```
            <div class="nums"><input id="beta_u_num" type="number"
step="0.1" value="1.0"><span class="rowhint">equality alignment
weight</span></div>
          </div>
        </label>

        <div class="section-gap"></div>

        <label class="row"><span>tau_hi in (0,1)</span>
          <div>
            <input id="tau_hi_range" type="range" min="0.01" max="0.99"
step="0.01" value="0.70">
            <div class="nums"><input id="tau_hi_num" type="number"
step="0.01" value="0.70"><span></span></div>
          </div>
        </label>
        <label class="row"><span>tau_eq in (0,1)</span>
          <div>
            <input id="tau_eq_range" type="range" min="0.01" max="0.99"
step="0.01" value="0.80">
            <div class="nums"><input id="tau_eq_num" type="number"
step="0.01" value="0.80"><span></span></div>
          </div>
        </label>

        <div class="section-gap"></div>

        <label class="row"><span>eps_a (clamp)</span>
          <div class="nums"><input id="eps_a_num" type="number" step="1e-7"
value="0.000001"><span class="rowhint">recommend 1e-6</span></div>
        </label>
        <label class="row"><span>V_eps (>0)</span>
          <div class="nums"><input id="V_eps_num" type="number" step="1e-
13" value="1e-12"><span class="rowhint">recommend 1e-12</span></div>
        </label>

        <pre class="mono">dv = (m_x - m_y) / max( |m_x| + |m_y| , V_eps )
du = u(a_x) - u(a_y)
s_gt(x,y) = tanh( beta_v*dv + lambda_u*du )
raw_eq = beta_v*|dv| + beta_u*|du|
s_eq(x,y) = 1 - 2*tanh(raw_eq)</pre>
      </div>
    </div>

    <div class="grid g2" style="margin-top:16px;">
      <div class="card">
        <div class="title">Optional gate (alignment-only)</div>
        <label class="row"><span>gate_used</span>
          <div><input id="gate_used_chk" type="checkbox" checked> <span
class="rowhint">alignment deltas are gated pre-score</span></div>
        </label>
        <label class="row"><span>g_t in [0,1]</span>
          <div>
            <input id="gt_range" type="range" min="0" max="1" step="0.01"
value="1.0">
            <div class="nums"><input id="gt_num" type="number" step="0.01"
value="1.0"><span></span></div>
          </div>
        </label>
        <pre class="mono">a_env = clamp_a( g_t * a_op , eps_a)
(Decision flow gates du and |du| before scoring.)</pre>
```

```
      </div>

      <div class="card" id="results_card">
        <div class="title">Results</div>
        <div class="nums"><div class="rowhint">dv</div><div class="mono"
id="dv_out"></div></div>
        <div class="nums"><div class="rowhint">u_x, u_y</div><div
class="mono" id="u_out"></div></div>
        <div class="nums"><div class="rowhint">du, |du| (gated)</div><div
class="mono" id="du_out"></div></div>
        <div class="nums"><div class="rowhint">s_gt(x,y)</div><div
class="mono" id="sgt_out"></div></div>
        <div class="nums"><div class="rowhint">s_eq(x,y)</div><div
class="mono" id="seq_out"></div></div>
        <div style="margin-top:10px; padding:10px; border:1px solid var(--
brd); border-radius:12px; background:#f1f5f9;">
          <div class="mono rowhint" style="font-size:12px; text-
transform:uppercase;">Banded report</div>
          <div style="font-size:18px; font-weight:700; margin-top:4px;"
id="report_out"></div>
          <pre class="mono" style="margin-top:8px">Mapping:
if s_eq >= tau_eq -> "equal"
else if s_gt >= +tau_hi -> "x > y"
else if s_gt <= -tau_hi -> "x < y"
else -> "undecided"</pre>
        </div>
      </div>
    </div>

    <div class="grid g2" style="margin-top:16px;">
      <div class="card">
        <div class="title">Manifest snippet (copy from here)</div>
        <textarea readonly id="manifest_out" class="mono"></textarea>
        <div class="footer">This snippet mirrors the current knobs and
inputs.</div>
      </div>
      <div class="card">
        <div class="title">Formulas (ASCII) & invariants</div>
        <pre class="mono">Clamp:      clamp_a(a, eps_a) = sign(a) *
min(|a|, 1 - eps_a)
Rapidity:    u(a) = atanh( clamp_a(a, eps_a) )
Greater-than:
  dv  = (m_x - m_y) / max(|m_x| + |m_y|, V_eps)
  du  = g_t * (u(a_x) - u(a_y))
  s_gt(x,y) = tanh( beta_v*dv + lambda_u*du )
Equality:
  raw_eq    = beta_v*|dv| + beta_u*g_t*|u(a_x) - u(a_y)|
  s_eq(x,y) = 1 - 2*tanh(raw_eq)
Banded mapping (precedence: equality first):
  if s_eq >= tau_eq -> "equal"
  else if s_gt >= +tau_hi -> "x > y"
  else if s_gt <= -tau_hi -> "x < y"
  else -> "undecided"
Bounds:
  |a| < 1 via clamp + tanh. dv is scale-invariant under c>0 rescaling of
magnitudes.</pre>
    </div>
    </div>

    <div class="footer">CC BY 4.0 — This demo is illustrative; see the spec
for full details.</div>
```

```
  </div>

<script>
  // Helpers
  const clampA = (a, eps) => { const s = Math.sign(a) || 1; const v =
Math.min(Math.abs(a), 1 - eps); return s * v; };
  const uOf = (a, eps) => Math.atanh(clampA(a, eps));
  const tanh = x => Math.tanh(x);
  const fmt = x => (!isFinite(x) ? String(x) : (Math.abs(x) !== 0 &&
(Math.abs(x) >= 1e6 || Math.abs(x) < 1e-6) ? x.toExponential(6) :
x.toFixed(6)));
  const el = id => document.getElementById(id);
  const toNum = (v, fb) => { if (typeof v === "number" && isFinite(v))
return v; if (typeof v !== "string") return fb; const s = v.replace(',',
'.').trim(); const n = Number(s); return isFinite(n) ? n : fb; };

  // Adjust spacer to fixed bar height
  function adjustBarSpacer(){
    const h = el("fixed_bar").offsetHeight || 64;
    el("bar_spacer").style.height = h + "px";
  }
  window.addEventListener("load", adjustBarSpacer);
  window.addEventListener("resize", adjustBarSpacer);

  const binds = [
    ["mx", 5], ["ax", 0.4], ["my", 3], ["ay", 0.1],
    ["beta_v", 1.0], ["lambda_u", 1.0], ["beta_u", 1.0],
    ["tau_hi", 0.70], ["tau_eq", 0.80],
    ["eps_a", 1e-6], ["V_eps", 1e-12],
    ["gt", 1.0]
  ];
  const state = {};
  function setBoth(name, v){ state[name] = v; const r = el(name+"_range");
if (r) r.value = String(v); const n = el(name+"_num"); if (n) n.value =
String(v); }

  // initialize values
  binds.forEach(([k, v]) => setBoth(k, v));
  el("gate_used_chk").checked = true;

  // Bind live updates (auto-compute)
  function bindAuto(idBase){
    const r = el(idBase+"_range"); const n = el(idBase+"_num");
    if (r) { r.addEventListener("input", e => { setBoth(idBase,
toNum(e.target.value, state[idBase])); update(true); });
             r.addEventListener("change", e => { setBoth(idBase,
toNum(e.target.value, state[idBase])); update(true); }); }
    if (n) { n.addEventListener("input", e => { setBoth(idBase,
toNum(e.target.value, state[idBase])); update(true); });
             n.addEventListener("change", e => { setBoth(idBase,
toNum(e.target.value, state[idBase])); update(true); }); }
  }
  binds.forEach(([k]) => bindAuto(k));
  el("gate_used_chk").addEventListener("change", () => update(true));

  function press(elm){
    elm.classList.add("is-pressed");
    setTimeout(() => elm.classList.remove("is-pressed"), 130);
  }

  function pulseResults(){
```

```
    const card = el("results_card");
    card.classList.remove("pulse-once");
    void card.offsetWidth;
    card.classList.add("pulse-once");
    const chip = el("mini_chip");
    chip.classList.remove("chip-pop"); void chip.offsetWidth;
chip.classList.add("chip-pop");
  }

  function update(doPulse){
    const mX = state.mx, mY = state.my, aX = state.ax, aY = state.ay;
    const betaV = state.beta_v, lambdaU = state.lambda_u, betaU =
state.beta_u;
    const tauHi = state.tau_hi, tauEq = state.tau_eq;
    const epsA = Math.max(1e-15, state.eps_a);
    const Veps = Math.max(1e-20, state.V_eps);
    const gUsed = el("gate_used_chk").checked;
    const gT = Math.min(1, Math.max(0, state.gt));

    const denom = Math.max(Math.abs(mX) + Math.abs(mY), Veps);
    const dv = (mX - mY) / denom;

    const ux = uOf(aX, epsA);
    const uy = uOf(aY, epsA);

    const du = (gUsed ? gT : 1) * (ux - uy);
    const d_u = Math.abs(du);

    const sgt = tanh(betaV*dv + lambdaU*du);
    const rawEq = betaV*Math.abs(dv) + betaU*d_u;
    const seq = 1 - 2*tanh(rawEq);

    let report = "undecided";
    if (seq >= tauEq) report = "equal";
    else if (sgt >= +tauHi) report = "x > y";
    else if (sgt <= -tauHi) report = "x < y";

    el("dv_out").textContent  = fmt(dv);
    el("u_out").textContent   = fmt(ux)+", "+fmt(uy);
    el("du_out").textContent  = fmt(du)+", "+fmt(d_u);
    el("sgt_out").textContent = fmt(sgt);
    el("seq_out").textContent = fmt(seq);
    el("report_out").textContent = report;

    const manifest = {
      V_eps: Veps, eps_a: epsA,
      beta_v: betaV, lambda_u: lambdaU, beta_u: betaU,
      tau_hi: tauHi, tau_eq: tauEq,
      gate_used: gUsed ? "on" : "off", g_t_demo: gT,
      demo_inputs: { m_x: mX, a_x: aX, m_y: mY, a_y: aY }
    };
    el("manifest_out").value = JSON.stringify(manifest, null, 2);

    const now = new Date();
    el("status").textContent = "Auto compute: on | Last updated: " +
now.toLocaleTimeString() + " ✓";
    el("mini_out").textContent = "s_gt=" + fmt(sgt) + ", s_eq=" + fmt(seq)
+ ", report=" + report;

    if (doPulse) pulseResults();
  }
```

```
  function resetDefaults(){
    const btn = el("reset_btn");
    press(btn);
    binds.forEach(([k, v]) => setBoth(k, v));
    el("gate_used_chk").checked = true;
    update(true);
    btn.blur();
  }

  // First render and spacer sizing
  update(false);
  (function adjust(){ const h = el("fixed_bar").offsetHeight || 64;
el("bar_spacer").style.height = h + "px"; })();

  // Prevent spacebar from scrolling page when body focused
  window.addEventListener('keydown', e => {
    if ((e.code === 'Space' || e.key === ' ') && document.activeElement ===
document.body) e.preventDefault();
  });
</script>
</body>
</html>
```

OMP