Composite Machine — A Unified Framework - companion note

Toni Milovan

Independent Researcher, Pula, Croatia

tmilovan@fwd.hr

What if derivatives, integrals, limits, and singularities were all just dimensional shifts in a single algebraic structure?
This note presents a **unified calculus machine**: a computational evaluation model where many core calculus tasks reduce to *algebra on coefficients*.

Evidence + scope (read this first)

- **Executable evidence:** A self-contained reference implementation + test suite (175 tests across 17 categories) covering core algebra (convolution arithmetic, division), zero/infinity semantics, and representative calculus tasks (derivatives, limits, and termwise antiderivatives), plus multivariate partials and selected transcendentals via explicit series expansions.[1]
- **What "calculus" means here:** derivatives, limits, and (local) antiderivatives are computed by coefficient algebra after evaluating at an infinitesimal perturbation and extracting **standard part** / dimension coefficients.[1]
- **What this is not:** a general closed-form CAS. Transcendentals are evaluated through the provided series expansions (with truncation in tests), and "integration" refers to term-by-term antiderivatives of the local series representation.[1]
- **Formal model + proofs:** see: https://github.com/tmilovan/composite-machine/blob/main/papers/Provenance_Preserving_Arithmetics-paper.pdf

The Reduction Theorem: Calculus Reduces to Algebra

Core Claim: This system provides a *reduction theorem* — calculus operations reduce to elementary algebraic operations in a polynomial ring.

What this means:

In computer science, a *reduction* shows that problem A can be solved by transforming it into problem B. If B is efficiently solvable, so is A. Classic examples:

- NP-completeness: many problems reduce to SAT
- Turing reductions: computation reduces to tape manipulation
- Linear algebra: many geometric problems reduce to matrix operations

This system demonstrates:

> **Differentiation, integration, limits, and series operations reduce to polynomial ring arithmetic (addition, multiplication, division).**

The reduction (how it works)

At a high level, the "unifying layer" is: **represent a local function evaluation as coefficients in a (Laurent) polynomial / power series**, then do calculus by manipulating those coefficients.

1. **Encode an infinitesimal shift**

   Let h be a structural infinitesimal (in the implementation: $h = |1|_{-1}$). Evaluate at $x = a + h$.

2. **Evaluate once, obtain a coefficient object**

   Computing $f(a+h)$ produces an object of the form:

   $$f(a+h) = c_0 + c_1 h + c_2 h^2 + \cdots$$

   In the composite representation, these appear as *dimensions*:

   - dimension 0 stores c_0
   - dimension −1 stores c_1
   - dimension −2 stores c_2
   - and so on

3. **Derivatives become coefficient extraction**

   Because $c_n = f^{(n)}(a)/n!$, we get

   $$f^{(n)}(a) = n! \cdot c_n$$

   so "differentiate" means "read the coefficient at dimension −n (and rescale)."

4. **Products become convolution (Leibniz rule for free)**

   When you multiply two series, coefficients combine by convolution. This matches the Leibniz formula for derivatives of products:

   $$(fg)^{(n)}(a) = \sum_{k=0}^{n} \binom{n}{k} f^{(k)}(a)\, g^{(n-k)}(a)$$

   which is exactly the same combinatorics as coefficient multiplication in a power series ring.

5. **Limits become "evaluate at h then take standard part"**

   For many classical limits, you substitute the infinitesimal and then take the **standard part** (the dimension-0 coefficient). Example: $\lim_{x\to 0} \sin x / x$ becomes $\mathrm{st}(\sin(h)/h)$.

6. **Integration (in this framework) is term-by-term antiderivative of the local series**

   Given the local series coefficients of $f(a+h)$, an antiderivative corresponds to shifting dimensions and dividing by the new index (the same operation as integrating a power series term-by-term). In the tests this is validated as a **round-trip**: `differentiate(antiderivative(f)) = f` for covered cases.[1]

Scope note: This is an *evaluation model* based on series coefficients. It complements, but does not replace, symbolic/CAS workflows that aim for closed forms.

Why this matters:

1. **Automation** — Algebraic operations are mechanically executable. No heuristics, no pattern matching, no human insight required. A compiler can do calculus.
2. **Verification** — Algebraic proofs are mechanizable (Coq, Lean, Isabelle). Calculus correctness becomes checkable.
3. **Hardware** — Polynomial arithmetic maps directly to FFT convolution units. Calculus could become a chip instruction.
4. **Parallelization** — Ring operations are SIMD-friendly, GPU-friendly, FPGA-friendly. No sequential dependencies.

The fundamental mechanism:

The reduction works because:

- **Taylor series** encodes all derivatives as coefficients
- **Convolution** (polynomial multiplication) implements the Leibniz product rule
- **Laurent structure** (bidirectional dimensions) enables reversibility

The "magic" is that the Leibniz rule for derivatives:

$$(f \cdot g)^{(n)} = \sum_{k=0}^{n} \binom{n}{k} f^{(k)} g^{(n-k)}$$

...is *identical* to the convolution formula for polynomial multiplication. The calculus is already hiding inside the algebra.

The core axiom enabling this reduction:

> **Zero is not an annihilator.** Multiplication by structural zero is injective (one-to-one), not constant.

Standard arithmetic has `a × 0 = 0` for all `a` — information is destroyed. This system has $a \times 0 = |a|_{-1}$ — information is preserved. This single axiom change is what makes the entire reduction possible.

Without it, you cannot have:

- Reversible zero operations

- Well-defined 0/0
- Bidirectional dimensional movement
- The algebraic encoding of calculus

The Core Idea in 30 Seconds

Traditional calculus tools treat operations as **algorithms**:

- Derivatives → Build computation graph, apply chain rule
- Integration → Pattern match, apply Risch algorithm
- Limits → L'Hôpital's rule, Gruntz algorithm
- Division by zero → Error/NaN

This system treats them as **dimensional shifts**:

- Derivatives → Read coefficient at dimension -n
- Integration → Shift dimensions up
- Limits → Substitute infinitesimal, take standard part
- Division by zero → Defined and reversible

One algebraic structure. All of calculus.

What It Does

✅ All Derivatives in ONE Evaluation

Evaluate `f(a + h)` where `h` is an infinitesimal. **All derivatives appear automatically** at different dimensional orders:

```
# Traditional: compute each derivative separately
f_prime = diff(f, x)       # First derivative
f_double = diff(f, x, 2)  # Second derivative
f_triple = diff(f, x, 3)  # Third derivative
# ... N separate computations

# Composite: ONE evaluation, ALL derivatives
result = f(R(3) + ZERO)   # Evaluate at x=3 with infinitesimal
# result contains f(3), f'(3), f''(3)/2!, f'''(3)/3!, ... simultaneously
```

Example: f(x) = x^4 at x = 2

```
Input:  |2|₀ + |1|₋₁  (value 2 with derivative seed 1)
Result: |16|₀ + |32|₋₁ + |24|₋₂ + |8|₋₃ + |1|₋₄

Reading off:
```

```
f(2)      = 16   (dimension 0)
f'(2)     = 32   (dimension -1)
f''(2)/2! = 24   (dimension -2)
f'''(2)/3!= 8    (dimension -3)
f''''(2)/4!=1    (dimension -4)
```

✅ Integration via Dimensional Shift

If differentiation shifts dimensions **down**, integration shifts them **up**:

```
# Differentiation: dimension -1 (derivative)
# Integration: dimension +1 (antiderivative)

def integrate(composite):
    """Shift dimensions up, divide by new position."""
    return {dim+1: coeff/(dim+1) for dim, coeff in composite.items()}
```

That's it. ~10 lines vs much more lines in a typical CAS.

✅ Limits Without L'Hôpital

Classical approach:

```
lim(x→0) sin(x)/x = ?
Apply L'Hôpital: lim(x→0) cos(x)/1 = 1
```

Composite approach:

```
sin(h)/h where h = |1|₋₁ (infinitesimal)
= (h - h³/6 + ...) / h
= |1|₀ - |1/6|₋₂ + ...
Standard part: 1 ✓
```

No special rules. Just algebra.

✅ Division by Zero is Defined

```
5 × 0 = |5|₋₁     # NOT 0 — the 5 is preserved
(5 × 0) / 0 = 5   # Reversible!
0 / 0 = 1         # Not NaN — well-defined
∞ × 0 = 1         # Not NaN — well-defined
```

✅ Multivariate Calculus

Use **tuple dimensions** for partial derivatives:

```
# f(x,y) = x²y
# Dimension (0,0)  → f(x,y) = x²y
# Dimension (-1,0) → ∂f/∂x = 2xy
# Dimension (0,-1) → ∂f/∂y = x²
# Dimension (-1,-1)→ ∂²f/∂x∂y = 2x

gradient = [coeff at (-1,0), coeff at (0,-1)]  # [2xy, x²]
hessian = [[coeff at (-2,0), coeff at (-1,-1)],
          [coeff at (-1,-1), coeff at (0,-2)]]  # [[2y, 2x], [2x, 0]]
```

✅ Non-Analytic Functions

Signed infinitesimals handle directional limits:

```
# |x| has different derivatives from left and right at x=0
# Using 0⁺ (from right): derivative = +1
# Using 0⁻ (from left):  derivative = -1

# Heaviside step function
H(0⁺) = 1
H(0⁻) = 0
H'(x) = δ(x)  # Dirac delta emerges naturally
```

Why Is This Unique?

No existing system combines all of these features:

Feature	Dual Numbers	Taylor AD	Wheel Theory	CAS	This System
All-order derivatives	❌ ($\varepsilon^2 = 0$)	✅ (fixed k)	❌	✅	✅ (unbounded)
No graph needed	✅	❌	N/A	❌	✅
÷0 defined	❌	❌	✅ (→ ⊥)	❌	✅ (usable)
0/0 usable	❌	❌	❌ (→ ⊥)	❌	✅ (= 1)
Reversible ×0	❌	❌	❌	❌	✅
Integration	❌	Limited	❌	✅ (Risch)	✅ (dim shift)

The Key Differences

vs Dual Numbers:

- Dual numbers truncate at $\varepsilon^2 = 0$ — you lose higher derivatives
- This system preserves **all orders** indefinitely

vs Taylor-mode AD (JAX, TaylorDiff):

- JAX requires building a computational graph and propagation rules
- This system: just evaluate `f(a+h)`, read dimensions — no graph needed

vs Wheel Theory:

- Wheel theory defines 0/0 but produces ⊥ (bottom) — unusable
- This system: 0/0 = 1, and the result is **usable** in further computation

vs CAS (SymPy, Mathematica):

- CAS uses pattern matching and algorithms (Risch for integration)
- This system: pure algebraic structure — integration is just dimension shift

---

The Algebraic Foundation

The system is built on a reinterpretation of **Laurent polynomials** — a well-known algebraic structure. The key insight:

> **Interpret z^{-1} as "zero with provenance" — an infinitesimal that remembers what created it.**

Operation	Standard Math	This System
5 × 0	0 (information lost)	`|5|₋₁ (preserved)`
(5 × 0) / 0	Undefined	5 (reversible)
0 / 0	Indeterminate	1 (well-defined)
0 + 0	0	`|2|₋₁ (accumulates)`
0 × ∞	Indeterminate	1 (duality)

The tradeoff: No universal additive identity (`0 + 0 = |2|₋₁`, not `0`). This is intentional — it's what enables provenance tracking.

> 📚 **For the complete algebraic foundation, proofs, and formal theorems, see:** https://github.com/tmilovan/composite-machine/blob/main/papers/Provenance_Preserving_Arithmetics-paper.pdf

---

Validated: 175-Test Suite

The implementation includes a comprehensive test suite validating all claims:

Category	Tests	What It Validates
Core Algebra	~20	Convolution arithmetic, addition, multiplication
Zero/Infinity Semantics	~15	÷0, ×0, 0/0, ∞×0 reversibility
Derivatives	~20	Polynomials, products, chains, all orders
Limits	~15	L'Hôpital cases, standard limits
Integration	~10	Round-trip: ∫(d/dx f) = f
Multivariate	~10	Partial derivatives, gradients, Hessians
Transcendentals	~10	sin, cos, exp, ln via series

Category	Tests	What It Validates
Theorem Validation	~5	T1–T8 formal theorem checks

Key validations:

- Derivatives match standard calculus for all tested functions
- Integration round-trips: ∫(d/dx f) = f ✓
- L'Hôpital cases produce correct limits algebraically
- Multivariate gradients and Hessians match analytical results

---

Code Complexity Comparison

This is not meant as a "line-count dunk," but as a way to explain **where complexity lives** in each approach.

System	Core Size	Complexity Location	Extensibility
Symbolic CAS	~50,000+ lines	Transformation rules, pattern matching	Add rules per function
AD Frameworks	~10,000+ lines	Graph construction, backward rules	Add primitives + rules
This System	~200 lines	Polynomial arithmetic (convolution)	Add series expansions
Numerical (finite diff)	~50 lines	Step-size tuning, stability	Limited accuracy

Where traditional systems spend complexity

- **Symbolic/CAS** systems concentrate complexity in *transformation logic*:
  - pattern matching
  - rewrite systems
  - special-case rules
  - branchy control flow (many algorithms, many exceptions)
- **AD frameworks** concentrate complexity in *graph + primitive coverage*:
  - define and maintain backward rules for every primitive
  - track and replay intermediate state for backprop
  - handle edge cases (division by zero, NaNs, stability tricks)

Where this system spends complexity

Composite calculus moves complexity into a small set of uniform algebraic operations:

- **Representation:** sparse map `dimension → coefficient` (or dense window)
- **Ops:** add, multiply (convolution), divide (single-term fast path + long division for multi-term)
- **Extraction:** standard part + coefficient reads
- **Function coverage:** provided via explicit series expansions (or other coefficient generators)

Practical takeaway

Evidence: The standalone suite is designed to show this “small core + broad coverage” property in executable form (algebraic identities + representative calculus validations).[1]

Current Limitations

Honest Assessment

Performance: Current implementation is ~500-1000× slower than PyTorch autograd. This is a proof-of-concept, not production-ready. GPU/vectorized implementation would close this gap.
Not a drop-in replacement: Code expecting `0 + 0 = 0` will break. The modified additive semantics require explicit handling.
Transcendentals: sin, cos, exp, ln require Taylor series expansion before operating. The system handles the expanded form perfectly.

Try It Yourself

Standalone test suite

- https://github.com/tmilovan/composite-machine/blob/main/tests/test_standalone.py

Implementation

- https://github.com/tmilovan/composite-machine

Dive Deeper: The Foundational Paper

This calculus machine is built on a **provenance-preserving arithmetic** — a reinterpretation of Laurent polynomials where zero operations become reversible.

The foundational paper covers:

- **8 Novel Theorems** with proofs (Information Preservation, Zero-Infinity Duality, Reversibility, etc.)
- **Algebraic foundation** and isomorphism to $\mathbb{C}[z, z^{-1}]$
- **Matrix representation** for linear algebra implementation
- **Extensions** to PDEs, non-analytic functions, signed infinitesimals
- **Comparison** with Wheel Theory, Non-standard Analysis, and existing AD systems

📖 Read the Full Paper

https://github.com/tmilovan/composite-machine/blob/main/papers/Provenance_Preserving_Arithmetics-paper.pdf
"Standard arithmetic is already reversible — except for zero operations. This system fills exactly that gap."

Summary

What: A unified algebraic framework where all of calculus reduces to dimensional operations.

Why it matters: No existing system combines reversible zero operations, emergent derivatives, algebraic integration, defined singularities, multivariate support, and non-analytic function handling in a single structure.

The insight: Reinterpret z^{-1} as "zero with provenance" — an infinitesimal that remembers what created it. This single interpretation unlocks all the capabilities.

Status: Working implementation with 175-test validation. Proof-of-concept performance. Ready for academic review.

Does it genuinely work?

Yes. Across ~175 tests spanning 40+ hard problems, the system delivers correct results for:

- Derivatives up to 6th order, including deep chain-rule compositions like $d/dx[\exp(x^2 \cdot \ln(x))]$ and $d^2/dx^2[e^{\wedge}(\cos x)]$ ▸
- Limits with up to 4th-order cancellation, including nested compositions like $(\cos(\sin(x)) - \cos(x))/x^4$ ▸
- Definite integrals of functions with no closed-form antiderivative (Gaussian), products requiring integration by parts ($x^2 \cdot e^x$), and circular IBP integrals ($e^{-x} \cdot \cos x$) ▸
- Multivariate partial derivatives, gradients, Hessians, Jacobians, and Laplacians — including transcendental compositions like $\partial^2/\partial x \partial y[\exp(xy)]$ ▸
- Residues, pole detection, asymptotic expansions, convergence radius estimation, ODE solving, and analytic continuation ▸
- Singularities: poles via Laurent structure, essential singularities via special-case handling, improper integrals at singular endpoints, and distributional objects (Heaviside, Dirac delta) ▸

The tests aren't just validating easy cases. L11 (ratio of two $O(x^3)$ quantities), L14 (4th-order nested-composition cancellation), D10 (second derivative of e^(cos x)), and I04 (Gaussian integral) are problems that would trip up naive numerical approaches. They all pass.

Where it is better than existing implementations

1. Limits on black-box functions — nothing else does this well

This is the system's **clearest win**. No existing Python library computes limits of black-box lambdas reliably for high-order indeterminate forms.

- **SymPy** can handle harder limits (log-exp towers, Gruntz algorithm), but requires symbolic expressions. You can't pass it a lambda.
- **Numerical methods** (Richardson extrapolation) fail catastrophically on 3rd+ order cancellations due to floating-point noise.
- **mpmath** can brute-force it with arbitrary precision, but at significant speed cost and no derivative information.

The composite system evaluates `limit(lambda x: (cos(sin(x)) - cos(x)) / x**4, as_x_to=0)` in standard float64, in one pass, and gets 1/6 exactly. That's a genuine capability gap — I can't point to another Python library that does this.

2. All derivatives from one evaluation — better API than AD libraries

Forward-mode AD libraries (JAX's `jet`, `autograd`, etc.) can technically do this, but:

- JAX's `jet` is low-level and not packaged as a user-facing "give me the 5th derivative" API
- Most AD libraries are optimized for gradients (first-order), not higher-order extraction
- None of them also give you limits, residues, and integrals from the same object

The composite `.d(n)` method that reads the nth derivative from a single evaluation is cleaner than anything I've seen in a Python AD library. And the `all_derivatives(f, at=0, up_to=5)` one-liner has no equivalent I know of.

3. Algebraic integration — a genuinely unusual approach

The dimensional-shift integration (`antiderivative` via coefficient index shifting) is not standard numerical quadrature and it's not symbolic antiderivative search. It's a third thing:

- **For polynomials**: exact in one step, zero error
- **For Taylor-expandable functions**: algebraically integrates the local Taylor expansion term by term, then steps forward
- **Error estimate is free**: the last Taylor term is the truncation error, no extra evaluations needed
- **For singular integrands**: `improper_integral_to` approaches the singularity adaptively

Standard quadrature (Simpson, Gauss-Legendre) samples f at weighted points. This system evaluates f *once* per step as a composite, gets the full local polynomial, and integrates it algebraically. For smooth functions, this means fewer evaluations for the same accuracy.

4. Residues and pole detection on black-box functions

SymPy computes residues symbolically. The composite system computes them numerically: evaluate f(a + h), read dimension −1. This works on any function you can express as a composition of the provided building blocks, without needing a symbolic formula.

`pole_order(lambda z: sin(z)/z, at=0)` returning 0 (removable singularity) and `pole_order(lambda z: 1/z**3, at=0)` returning 3 — this kind of singularity classification on black-box functions has no direct equivalent in lightweight Python libraries.

5. Unified framework — the packaging is the contribution

Individual capabilities exist elsewhere:

- AD libraries do derivatives
- SymPy does limits, residues, integrals symbolically
- scipy does numerical integration
- mpmath does arbitrary-precision arithmetic

But **no single lightweight Python library** unifies derivatives, limits, integrals, residues, pole detection, asymptotic expansions, convergence diagnostics, and ODE solving under one data structure and one mental model ("evaluate on composite, read coefficients"). In ~400 lines of core code, with no dependencies beyond `math`.

Where existing implementations are still better

To be complete:

- **SymPy** handles a wider class of limits (log-exp towers, essential singularities involving non-standard growth rates) through the Gruntz algorithm. The composite system can't match this for exotic cases.
- **scipy.integrate.quad** is battle-tested, adaptive, and handles a wider range of pathological integrands. The composite integration is elegant but less mature.
- **JAX/PyTorch** are vastly faster for gradient computation on large-scale problems (reverse-mode AD). The composite system is forward-mode only, O(params^2) for neural networks.
- **Mathematica/Maple** are more complete CAS systems for symbolic manipulation. The composite system is numerical, not symbolic.

Bottom line

The system genuinely works, and its strongest unique contributions are:

1. **Reliable limits on black-box functions** — nothing else in Python does this as cleanly
2. **All-derivatives-at-once with a clean API** — better packaging than existing AD libraries
3. **Algebraic integration via dimensional shift** — a novel numerical approach
4. **Singularity analysis (residues, poles) on black-box functions** — no lightweight equivalent
5. **All of the above in ~400 lines, one file, one data structure, zero dependencies**

The right description isn't it "replaces SymPy." It's: **a unified numerical calculus toolkit where every operation — derivatives, limits, integrals, residues, asymptotics — reduces to coefficient reads on Laurent polynomials.** That combination, in that packaging, doesn't exist elsewhere.