Floating Point Is Broken

Here's What We're Replacing It With

Technical Article

The Problem No One Questions

Floating point was not adopted because it was ideal—it was adopted because it matched the constraints of early hardware.

In the first generation of computing systems, memory was scarce, hardware was limited, and precision was expensive. Floating point offered a compact, lossy encoding scheme for approximating real numbers within a fixed number of bits. It was a compromise that proved scalable.

That compromise became the default. Floating point arithmetic remains the foundation of nearly all modern computation—and the assumptions behind it are rarely re-examined.

For example, consider the input:

```
sqrt(2)
```

This is not stored as a structural object. It is immediately evaluated:

```
1.4142135...
```

The original form is lost. Only a flattened numerical approximation remains.

In conventional systems, evaluation is assumed to be immediate. Structure is discarded by default, and symbolic identity is erased at the point of entry. Floating point is fast, portable, and sufficient for many numerical tasks, but fundamentally lossy:

- Rounding errors occur by default
- Resolution is constrained by bit width
- Irrational values are truncated into decimal approximations
- Symbolic expressions—including functions—are evaluated and flattened

These constraints are not anomalies—they are built into the foundation. Every layer built on top of floating point inherits the same behavior, reinforcing the same logic throughout the system. This recursive pattern—evaluating early, discarding form, and collapsing identity—is mirrored from numerical encoding all the way up to high-level computation.

How Symbolic Computing Works—and What It Makes Possible

The number is already structured. It does not need to be approximated or reduced. It can be preserved.

As seen earlier, the expression `sqrt(2)` is typically evaluated immediately, producing a truncated decimal. In symbolic computing, it can be preserved exactly as `2**(1/2)`, where exponentiation is retained structurally. Each component—`2`, `1/2`, and the operator `**`—is stored as an Independent Symbolic Unit (ISU). Together, they form an Independent Symbolic Expression (ISE) that remains unevaluated, accessible, and fully reusable.

Classical systems store the result of a computation. Symbolic systems store the computation itself.

What follows are examples of how preserving symbolic form changes the behavior of computation. When expressions are retained rather than resolved, new capabilities emerge: reversibility, introspection, and compositional control.

- `L2(x)` denotes the logarithm base 2 of `x`. In most systems, this is immediately evaluated into a floating-point approximation. In symbolic computing, the form `L2(x)` remains available as a compositional object. It can be reused, transformed, or passed into other symbolic expressions. The expression is stored symbolically, not reduced.
- In programming, consider a function definition: `f(x) = x × 2`. A call like `f(3)` is typically evaluated immediately, producing `6`. The original expression is lost, along with the ability to reference or transform the function call itself. In symbolic computing, `f(3)` is retained as a symbolic expression and stored structurally. It can be composed with other functions (such as `g(f(3))`), reused in new contexts, or reversed to solve for its inputs. The output remains available, but the form is preserved.
- In data pipelines, a sequence like `df.filter(...).groupby(...).sum()` is typically executed step by step, with each stage overwriting the last. The end value is computed, but the structure of the pipeline is discarded. In symbolic computing, the entire transformation is preserved as a symbolic expression (ISE). Rather than storing the full expression repeatedly, it can be assigned a symbolic identifier (ISU), allowing the pipeline to be compressed, reused, or referenced across contexts. Because the symbolic expression is stored unevaluated, the structure remains accessible across all operations.

In symbolic computing, output remains fully accessible. In all of the examples above, computations yield results just as they do in classical systems. The key difference is that evaluation is not automatic: it occurs only when explicitly requested or when structurally required by the symbolic form.

The tendency to evaluate immediately reflects a paradigm that no longer serves us. That constraint made sense in a time when memory was scarce and hardware was limited. But that context has changed. What remains is the logic we inherited: expressions must resolve, results must be stored as numbers, and structure is not preserved. We continue to follow this model not because it's needed, but because it's embedded in how we think about computation.

Symbolic Computing in Practice

The following examples show how symbolic computing behaves in real-world contexts—replacing approximation with preserved form, and enabling new execution models across domains.

Eliminating Floating Point Errors

One of the most well-known examples of floating-point failure is comparison logic:

```
(0.1 + 0.2) == 0.3    # False
```

Numbers like `0.1` and `0.2` cannot be represented exactly in binary floating-point form. The result of `0.1 + 0.2` is actually something like `0.30000000000000004`, which fails a direct equality check. Workarounds exist—such as comparing within a tolerance—but they are fragile and context-dependent. The deeper issue is that structure is lost before computation even begins.

In symbolic form:

```
1/10 + 2/10 == 3/10    # Symbolically exact
```

The structure is preserved. No rounding occurs. Equality holds. Rather than converting values into binary approximations, symbolic computing stores the expression exactly—as a structured ISE composed of ISUs. The comparison operates on symbolic form, not on floating-point estimates.

This is not just a fix for a single edge case. It addresses an entire class of problems inherent to floating-point arithmetic: underflow, overflow, truncation, rounding drift, representation gaps, and loss of symbolic form.

Symbolic computing takes a different approach: values retain their structure throughout computation. Evaluation is deferred—contextual, symbolic, and only applied when explicitly required.

Symbolic Cryptography

Cryptographic systems like RSA rely on operations involving large prime numbers and exponentiation. A typical encryption step looks like:

$$c = m^e \mod n$$

This expression means that the message m is raised to the power of e, and the result is computed modulo n. In practical terms, `mod n` means "take the remainder after dividing by n." Modular exponentiation applies this operation step by step, reducing intermediate results throughout the computation to prevent overflow. This avoids storing large numbers—but it is computationally expensive. As a result, the size of the exponent e must remain within practical limits. If the number becomes too large, the time required to compute or store it quickly becomes a bottleneck.

However, the need to reduce the result c is not intrinsic to the operation—it reflects the logic that governs all of computing: expressions must resolve, and results must be stored as numbers.

Symbolic computing changes this. Rather than reducing or evaluating the expression, it preserves it symbolically:

```
c = m**e
```

(Symbolic representation of exponentiation: m^e)

This expression is not computed in advance—it is stored directly in memory as a symbolic object. The structure remains fully intact and accessible throughout computation.

Each component—m, e, and c—is treated as an ISU. The composed expression $m * e$ is an ISE, and the symbolic assignment $c = m * *e$ is preserved in inline form, unevaluated.

Because the expression is not evaluated or reduced until explicitly required, storing it is nearly instantaneous. There is no modular reduction, no overflow management, and no scaling penalty. The size of the expression is no longer a constraint. Whether the exponent is 65537 or a thousand times larger, it can be represented symbolically as a compositional object built from ISUs and ISEs. These expressions are stored in symbolic memory as graphs—referenced, restructured, and reused without loss or delay. Evaluation may still occur at the final stage, such as when generating ciphertext. The result may be a very large number, but symbolic computing does not require that number to be stored—only the expression that produces it.

Symbolic Scientific Computing

Scientific computing depends on structured mathematical operations: differentiation, integration, transformation, and recursion. In classical systems, these operations are performed numerically. Structure is discarded early, and functions are flattened into approximations.

Consider the derivative of a simple function:

$$f(x) = \cos(x) \quad \Rightarrow \quad f'(x) = -\sin(x)$$

In a classical system, this is typically approximated using finite differences or a truncated Taylor series:

$$f'(x) \approx -x + \frac{x^3}{6} - \frac{x^5}{120} + \dots$$

This approximation must be recomputed for every input or recomposed function. If we differentiate again, the entire structure is re-expanded. Symbolic identity is lost at each step, and the cost of evaluation increases.

In symbolic computing, the transformation is preserved exactly.

```
d/dx [c(x)] = -s(x)
```

No expansion is performed. No approximation is introduced. The result is a new ISE, composed of ISUs and nested ISEs—such as `c(x)` and `s(x)`—that remains fully symbolic and structurally intact. It can be reused, composed, or transformed—without ever being flattened.

Symbolic scientific computing allows expressions to evolve without evaluation. Transformations remain symbolic, and results are stored structurally at every step. Evaluation can still occur—but only when explicitly requested, typically at the final stage. This enables exact computation throughout the process, with no structural loss until the moment of resolution.

Symbolic Audio

In traditional systems, audio is stored as a stream of floating-point samples—thousands of small numbers representing amplitude over time. A single tone, like the natural musical pitch A4 at 432 Hz, might be defined as:

$$f(t) = \sin(2\pi \cdot 432 \cdot t)$$

But instead of preserving that expression, most systems flatten it into a long sequence of approximated values, sampled at fixed intervals:

```
0.0, 0.309, 0.588, 0.809, 0.951, ...
```

Each value represents the amplitude of the wave at a specific point in time—such as $t = \frac{1}{44{,}100}$, $t = \frac{2}{44{,}100}$, and so on. The shape of the waveform is reconstructed from these discrete samples, but the structure is lost. The original expression is no longer present. The sound becomes data—decomposed, evaluated, and compressed.

In symbolic computing, that same tone can be stored directly in its original form:

```
f(t) = s(2*pi*432*t)
```

The expression is not evaluated. It is preserved as a symbolic object—exact, legible, and reversible. The variable t remains symbolic and continuous. Transformations such as pitch shifts, tempo changes, modulation, and layering can all be applied directly to the expression itself, without resampling or distortion.

While the example above represents a single tone, symbolic audio extends naturally to complex sound. Musical passages can be constructed as graphs of symbolic expressions—layered combinations of sine and cosine components, phase structures, and envelopes—each stored and transformed as symbolic form. These graphs can evolve over time and represent musical structure in its true form—as layered, generative expression rather than sampled data. This enables higher-quality recording and compositional control.

Because the structure is preserved rather than sampled, symbolic audio is naturally compact. A single expression—or a graph of expressions—can represent entire passages of sound, replacing thousands of numerical points without loss.

When played, the expression is evaluated only at the final stage, and only at the resolution required by the audio system. Until that moment, the symbolic form remains intact. Playback can adapt dynamically: the expression can be evaluated at finer time intervals if needed, enabling smoother output and higher fidelity.

Symbolic Matrix Computation

In traditional systems, matrix multiplication is performed numerically. Each entry is stored as a floating-point value, and all operations are flattened and evaluated step by step.

In symbolic computing, a matrix is represented as an ISE composed from ISUs—each unit pairing a value with a symbolic tag, such as its position. These units allow blocks, sub-blocks, or individual entries to be stored, transformed, or executed independently—without flattening.

In classical notation, a 2×2 matrix might be written simply as:

```
A = 1, 2; 3, 4
```

In symbolic computing, each scalar becomes a distinct ISU—its numerical value paired with a structural tag that encodes its location:

```
A = 1A11, 2A12; 3A21, 4A22
```

In this case, symbolic computing does something more than preserve form—it introduces structure. Classical matrix entries are anonymous numbers. Symbolic entries carry identity, position, and meaning. At first glance, this may seem like added complexity, but it

enables a new kind of resolution—symbolic granularity—that makes computation more precise, more efficient, and more controllable. Each layer of structure becomes a new point of control.

This symbolic resolution enables two distinct forms of parallelism:

Spatial Parallelism. Consider a 1000×1000 matrix—common in Artificial Intelligence workloads. In symbolic computing, this can be subdivided into 10 blocks of 100×100, which can themselves be subdivided again into smaller 10×10 matrices. Each submatrix is tagged with identity and structure, allowing every block to be processed independently and in parallel.

Even within a single block, symbolic tags allow entries to be grouped by type: rational numbers can be processed together, irrational entries grouped separately, and symbolic phases sorted or deferred depending on context. This enables phase-aware grouping and selective evaluation. Not all entries need to be evaluated at once, or in the same way.

Symbolic structure also supports reuse. In large-scale matrix applications—such as video processing or simulation—identical sub-blocks or repeated matrices often occur. Rather than recomputing or storing them multiple times, symbolic systems reference them structurally. Repeated components are stored once and reused across execution, forming a graph of compositional structure that is both memory-efficient and execution-ready.

Temporal Parallelism. In traditional systems, a sequence of matrix multiplications must be computed step by step in a fixed order:

$$A \times B \times C \times D \times E \times F$$

In symbolic computing, associativity allows the chain to be grouped and computed in parallel:

$$(A \times B), (C \times D), (E \times F)$$

Symbolic computing enables parallelism not just across data, but across time. In a classical system, a sequence of matrix multiplications must be computed in fixed order. In a symbolic system, subexpressions are tagged structurally and can be grouped, distributed, and reused. When operations repeat across a sequence, they are not recomputed—they are referenced symbolically. This reduces both execution time and memory cost.

Symbolic computing makes it possible to distribute computation across processors in whichever way the structure allows—spatially, temporally, or both. Large matrices can be subdivided; long multiplication chains can be partitioned. When symbolic form is preserved, execution strategy becomes flexible: the system can adapt to matrix size, sequence length, and available resources without flattening or recomputing anything.

ASN: The Notation Behind It

All of this is built on Adaptive Symbolic Notation (ASN)—a structured symbolic grammar that is compact, expressive, and compositional. At its core are Independent Symbolic Units (ISUs) and Independent Symbolic Expressions (ISEs). These are not representations of values—they are the structure of the computation itself.

Whether representing numbers, functions, transformations, or matrix operations, ASN encodes computation in compositional form. Expressions are stored structurally, unevaluated, and resolved only when needed.

Symbolic computing doesn't need to be deployed all at once. A single symbolic substitution, a tagged function, or a structure-preserving transformation is enough to begin—by saving it as a symbolic object rather than flattening it. That single shift is already a structural departure.

Where to Go From Here

To explore what symbolic computing makes possible in practice—including a full overview of the system architecture—visit:

`https://symboliccomputing.com`

If you're already on the site, continue here:

- **Floating Point vs Symbolic Notation:** `https://symboliccomputing.com/floating-point`
- **Symbolic Cryptography:** `https://symboliccomputing.com/use-cases/cryptography`
- **Matrix Computation:** `https://symboliccomputing.com/use-cases/matrix`
- **Scientific Computing:** `https://symboliccomputing.com/use-cases/scientific`
- **White Paper:** `https://symboliccomputing.com/whitepaper`