Asgard: A Programming Language for Dynamical Systems

Erik Arne Mathiesen-Dreyfus

Gimle Labs, Paris
www.gimlelabs.com

Abstract

We introduce Asgard, a programming language for dynamical systems. Around equations written in a LEAN-style notation, Asgard provides the full toolchain: a parser that produces typed compositional circuits, a rewrite engine that simplifies them through symbolic identities, a compiler that lowers them to JAX, and a runtime that executes them as fast, differentiable code. Unlike implicit approaches (Neural ODEs, PINNs) that learn black-box dynamics $\dot{x} = f_\theta(x)$, Asgard maintains explicit mathematical structure throughout: equations are parsed from a LEAN-style notation, compiled into signal-flow circuits built from atomic operations and categorical combinators, and evaluated under interchangeable calculi—deterministic, stochastic, or discrete—without changing the circuit itself. These circuits form a traced monoidal category, enabling algebraic rewriting, equivalence proofs, and formal composition. Black-box components allow learned functions (e.g., neural networks) to be embedded within otherwise explicit circuits, and end-to-end differentiation through feedback loops enables joint optimisation of structural parameters and learned components. We describe the architecture, the syntax-semantics separation, and the design choices that make explicit structure a practical alternative to implicit function approximation.

1 Introduction

The dominant paradigm for learning dynamical systems is *implicit*: a neural network approximates the dynamics as a black-box function $\dot{x} = f_\theta(x)$. Neural ODEs [4], Physics-Informed Neural Networks [12], and their extensions are powerful precisely because they sidestep the hard problem of structural identification in favour of function approximation.

But this flexibility comes at a cost. An implicit model cannot be inspected to understand *why* a system behaves as it does. Two learned subsystems cannot be composed through typed interfaces. The standard tools of control theory—Lyapunov analysis, reachability, formal verification—do not apply to black-box neural dynamics.

Classical system identification methods [3, 15] do recover explicit structure, but they require hand-crafted function libraries, struggle with hybrid or stochastic systems, and scale poorly.

Asgard takes a different approach: it is a programming language for dynamical systems in which the mathematical structure of the system is the primary object of computation. You write the governing equations in a LEAN-inspired notation, and Asgard provides the full toolchain around them: a parser that turns the notation into a typed intermediate representation—*circuits*—a rewrite engine that simplifies them through symbolic identities, a compiler that lowers them to JAX, and a runtime that simulates them as fast, differentiable code. These four stages—parse, rewrite, compile, simulate—carry a model from notation to simulation, with the typed circuit as the representation that everything else operates on. The structure stays explicit and typed all the way from equation to execution, visible and manipulable at every stage.

This note describes the architecture and core ideas. §2 covers the equation language. §3 introduces the typed circuit representation. §4 explains the interchangeable calculi. §5 describes the execution model. §6 covers hybrid systems, end-to-end optimisation, and coupled dynamics. §7 traces concrete examples through the full pipeline.

2 Syntax: A Language for Equations

Asgard uses a notation inspired by the LEAN proof assistant for writing equations. The surface syntax is purely structural—no numerical meaning is assigned at parse time.

Definition 1 (Equation syntax)**.** *An Asgard equation is a string over the grammar:*

$$\begin{aligned} \mathit{expr} ::= {} & \mathit{int}(\mathit{expr}, \mathit{id}) \mid \mathit{diff}(\mathit{expr}, \mathit{id}) \\ & \mid \mathit{sde}(\mathit{expr}, \mathit{expr}, \mathit{id}) \mid \mathit{sde}(\mathit{expr}, \mathit{expr}, \mathit{expr}, \mathit{id}) \\ & \mid \mathit{expr} + \mathit{expr} \mid \mathit{expr} * \mathit{expr} \mid \mathit{expr}^n \\ & \mid \mathit{sqrt} \mid \mathit{exp} \mid \mathit{log} \mid \mathit{sin} \mid \mathit{cos} \mid \lambda \mathit{id}.\mathit{expr} \mid \$\mathit{id} \mid \mathit{id} \mid \mathit{const} \end{aligned}$$

where `int` denotes integration, `diff` differentiation, and `sde` a stochastic process specified by its drift, diffusion, and state variable (the four-argument form adds a jump term for Lévy processes). Lambda abstractions (λ) and named parameters (`$`) support parameterised equations.

The grammar shown is a simplified excerpt. The full parser uses standard mathematical precedence ($+$ binds weaker than $*$, which binds weaker than ^; exponentiation is right-associative). Lambda abstractions are beta-reduced at the equation level before circuit compilation, so the circuit layer never sees them directly. Named parameters (`$mu`, `$sigma`, etc.) compile to `param`(n) atomics whose values are supplied at runtime.

For example, the ODE $f' = f$ with $f(0) = 1$ is written as `f = f0 + int(f, x)`. The parser produces a structured parse tree that can be normalised and rewritten before compilation. The rewrite engine encodes identities such as the fundamental theorem of calculus:

$$\texttt{diff}(\texttt{int}(f, x), x) \longrightarrow f$$

This simplifies equations at the symbolic level, before any numerics are involved.

3 A Typed Intermediate Representation

Compilation transforms an equation into a *circuit*—a typed signal-flow network built from two kinds of components.

3.1 Atomic operations

Atomics are computational primitives with fixed input/output types (degrees):

Table 1: Selected atomic operations and their type signatures

Category	Operation	$(d_{\text{in}}, d_{\text{out}})$	Semantics
Generators	`const`(c), `param`(n)	$(0, 1)$	Constant or named parameter stream
	`sin`(n), `cos`(n), `exp`(n)	$(0, 1)$	Transcendental coefficient streams
Transformers	`id`	$(1, 1)$	Identity (pass-through)
	`register`(x), `deregister`(x)	$(1, 1)$	Formal integration / differentiation
	`scalar`(s)	$(1, 1)$	Multiply by a scalar
	`var`(n)	$(1, 1)$	Variable reference (replaced by `id` during trace wiring)
Binary	`add`, `multiplication`	$(2, 1)$	Pointwise addition, Cauchy product
Routing	`split`	$(1, 2)$	Duplicate a signal
	`swap`	$(2, 2)$	Exchange two signals
	`terminal`	$(1, 0)$	Discard a signal
Stochastic	`stochastic_register`	$(1, 1)$	Wiener integral $\int \cdot\, dW$
	`jump_register`	$(1, 1)$	Poisson integral $\int \cdot\, dN$
External	`black_box`(n)	$(1, 1)$	Apply named external callable
	`black_box_source`(n)	$(0, 1)$	External callable with no input

3.2 Combinators

Combinators wire atomics into larger circuits:

- **Composition**: `composition`(f, g) feeds the output of f into the input of g. Requires $d_{\text{out}}(f) = d_{\text{in}}(g)$.
- **Monoidal product**: `monoidal`(f, g) runs f and g in parallel. Degrees add: $d_{\text{in}} = d_{\text{in}}(f) + d_{\text{in}}(g)$.
- **Trace**: `trace`(f) feeds the last output of f back to its first input, implementing a fixed-point / feedback loop. Degrees decrease by one: $(d_{\text{in}} - 1, d_{\text{out}} - 1)$.
- **Power series composition**: `compose`(f, g, x) computes $f(g(x))$ in coefficient space. Unlike the three combinators above, this is not part of the traced monoidal structure—it is a domain-specific operation for stream calculus evaluation.

3.3 The compilation pipeline

Compilation proceeds in six stages, each using the same rewrite engine:

1. **Normalisation**: apply algebraic identities (distributivity, associativity) to canonicalise the equation.
2. **Beta reduction**: eliminate lambda applications by substitution.
3. **Variable isolation**: identify bounded variables (those with higher derivative than integral degree) and rewrite to isolate them on the left-hand side. For example, $f'(x) = f(x)$ becomes $f = f_0 + \int f \, dx$.
4. **Equation-to-circuit translation**: rewrite rules map each equation construct to its circuit equivalent—`int` to `register`, `+` to `add`, identifiers to `var`, and so on.
5. **Degree fixing**: insert `terminal` operations where input degrees are mismatched (e.g., a `param` composed with a unary transformer).
6. **Trace wiring**: detect feedback variables (those that appear both as inputs and outputs), replace their `var` references with `id`, and wrap the circuit in `trace`. For coupled systems, cross-variable references are resolved by routing circuits built from `split` and `swap`.

3.4 Example: compiling an ODE

The equation $f = f_0 + \int f \, dx$ compiles to:

$$\texttt{trace}\Big(\texttt{composition}\big(\texttt{monoidal}(\texttt{var}(f_0),\ \texttt{composition}(\texttt{id},\ \texttt{register}(x))),\ \texttt{add}\big)\Big) \tag{1}$$

Reading inside-out: the feedback variable f enters via `trace`. It passes through `register`(x) (integration), runs in parallel with the initial condition f_0, and the two signals are summed by `add`. The output feeds back—implementing the recursive definition. Figure 1 illustrates the signal flow.

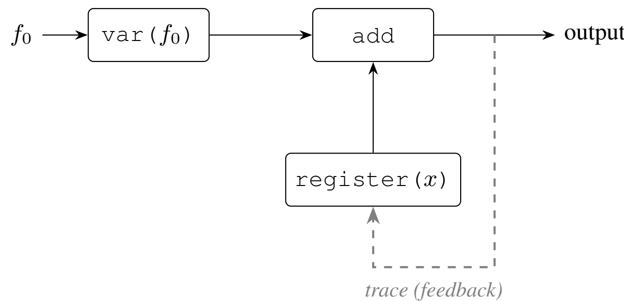

Figure 1: Signal-flow diagram for the circuit in Eq. (1). The dashed line is the trace (feedback loop): the output feeds back through `register` (integration), is combined with the initial condition f_0 via `add`, and the process repeats.

3.5 Categorical structure

These circuits form a *traced monoidal category* [6]. The idea of using traced monoidal categories as a semantic framework for computation with feedback was introduced in [10], where Hoare-style reasoning was lifted to an abstract categorical setting; the full soundness and completeness theory appears in [1]. Asgard inherits this structure as both a type system and a practical rewrite engine.

The most immediately practical consequence of this structure is the *type system*: every composition `composition`(f, g) requires $d_{\text{out}}(f) = d_{\text{in}}(g)$, and this is checked statically before any numerics run. Wiring errors are caught at compile time, not discovered as runtime shape mismatches.

Beyond type checking, the categorical axioms—associativity of composition, the interchange law for the monoidal product, naturality and vanishing for trace—constitute a *proof system* for circuit equivalence. Two circuits that can be connected by a chain of rewrites compute the same function. This enables:

- **Simplification**: reduce circuits to simpler equivalent forms. Examples of semantics-preserving rewrite rules:

$$\texttt{composition}(\texttt{id},\ f) = f \qquad \text{(identity law)}$$
$$\texttt{composition}(\texttt{scalar}(a),\ \texttt{scalar}(b)) = \texttt{scalar}(ab) \qquad \text{(scalar fusion)}$$
$$\texttt{composition}(\texttt{register},\ \texttt{deregister}) = \texttt{id} \qquad \text{(cancellation)}$$

- **Equivalence proofs**: show that two differently structured circuits compute the same function, without running them—by exhibiting a rewrite path between them.
- **Approximation**: rewrite a circuit into a structurally simpler form that is approximately equivalent, for instance by truncating higher-order terms or replacing subnetworks with known analytic expressions. Unlike the exact rewrites above, approximate rewrites are not justified by the categorical axioms alone—they require numerical validation (e.g., bounding the error via simulation).
- **Formal composition**: build complex models by wiring simpler circuits, with the type system guaranteeing that input/output degrees match at every connection point.

3.6 Closed-form solutions as trace-free circuits

An intriguing consequence of this rewrite system concerns closed-form solutions. A circuit containing `trace` represents a system defined by a fixed-point equation—an ODE, a recurrence, a feedback loop. Solving it amounts to eliminating the feedback: finding an equivalent circuit that uses only atomic operations, composition, and monoidal product, but *no trace*.

A trace-free circuit is, by definition, a direct computation: it maps inputs to outputs without iteration or recursion. It may use special functions (`exp`, `sin`, `sqrt`, etc.) as atomics, but it contains no feedback. This is precisely what we mean by a closed-form solution.

We conjecture that a closed-form solution to a dynamical system exists if and only if there is a valid rewrite—a proof, in the categorical proof system—from the original traced circuit to a trace-free equivalent. This reframes the classical question of "does this ODE have a closed-form solution?" as a question about the rewrite theory of traced monoidal categories: does a trace-elimination proof exist? The conjecture connects to classical results in differential Galois theory (which characterises when an ODE is solvable in closed form) and, via Bainbridge's duality between continuous and discrete traced monoidal categories [2], to the question of loop elimination in programs—where trace-elimination is known to be undecidable in general. We develop this connection in a companion note.

4 Semantics: One Circuit, Many Meanings

A compiled circuit is semantics-free—it describes the *structure* of a computation, not its numerical interpretation. To obtain a result, one selects a *calculus*. Asgard provides three:

Table 2: Interchangeable calculi and their interpretations

Calculus	`register` →	Streams →	Use cases
Real	Formal integration	Taylor coefficients	Deterministic ODEs
Stochastic	Itô/Stratonovich integral	Multi-index expansion	SDEs, Monte Carlo
Discrete	Unit delay	Sequence terms	Recurrences, Markov chains

The same circuit structure that solves the exponential ODE under **RealCalculus** can, under **DiscreteCalculus**, generate the Fibonacci sequence—`register` becomes a unit delay, and `multiplication` becomes a Cauchy product (convolution of sequences).

This separation has a practical consequence: one can develop and validate a model's structure deterministically, then switch to stochastic execution to study noise effects—without changing the model itself.

5 Execution Model

Circuits compile to JAX programs. The execution strategy depends on the circuit's structure and the chosen calculus. This is a central point: the *same* circuit can be executed under different numerical schemes, each providing different trade-offs between accuracy, speed, and the class of systems it handles. The choice of execution scheme is a semantic decision, entirely orthogonal to the syntax of the circuit.

5.1 Stream calculus

For deterministic ODEs, Asgard can solve by computing Taylor coefficients via *Picard iteration* in coefficient space, following Rutten's stream calculus [14].

The solution is represented as a stream of coefficients $[c_0, c_1, c_2, \ldots]$. The `register` operation shifts coefficients right and divides by index—the power series analogue of integration. The `trace` combinator implements iteration: feed an initial guess through the circuit, use the output as the next guess, repeat until convergence.

For $f' = f,\ f(0) = 1$, iteration yields $c_n = 1/n!$—the Taylor coefficients of e^x—the exact Taylor coefficients of e^x. (The coefficients are computed exactly; it is the series reconstruction that has a finite convergence radius determined by the singularity structure of the solution.) This approach is JIT-compilable as a single JAX loop, naturally handles nonlinear systems, and produces the full analytic structure (all coefficients) rather than trajectories at discrete time points.

5.2 Time-stepping integrators

Circuits can equally be executed via classical time-stepping methods, where the circuit encodes the right-hand side of the differential equation. All stochastic schemes support both Itô and Stratonovich interpretations; the Stratonovich variants use midpoint corrections (Heun predictor-corrector) to preserve the ordinary chain rule.

Table 3: Available time-stepping integrators (deterministic and stochastic)

Method	Order	Notes
Runge–Kutta (RK4)	4.0	Classical fixed-step method. Four function evaluations per step.
Dormand–Prince (RK45)	5.0	Adaptive step-size via embedded 4th/5th-order pair. Automatically adjusts step size to meet a user-specified error tolerance.
Euler–Maruyama	0.5	Single evaluation per step. Default for SDEs. Stratonovich variant uses Heun predictor-corrector.
Milstein	1.0	Uses JAX autodiff for the diffusion derivative $\partial\sigma/\partial x$. Exact for geometric Brownian motion.
SRK1 (Platen)	1.0	Derivative-free: achieves Milstein-order accuracy via a finite-difference approximation with two function evaluations. Works with non-differentiable diffusions.

The same circuit can be run under RK4 for deterministic exploration, RK45 for adaptive accuracy, or any of the stochastic schemes for noise-driven systems—without any change to the circuit itself.

5.3 Stochastic Taylor expansion

A third execution path combines stream calculus with Monte Carlo sampling. Deterministic coefficients c_α are computed once via Picard iteration over a multi-index tree [8], where each index encodes an iterated integral against dt, dW, or dN (for jump processes). Each Monte Carlo path then samples the iterated integrals $I_\alpha(t,\omega)$ and evaluates $X(t,\omega) = \sum_\alpha c_\alpha\, I_\alpha(t,\omega)$. Because the coefficients are shared across all paths, this is particularly efficient for large ensembles. Antithetic variates—generating negatively correlated Brownian paths—are available as a variance reduction technique.

For long time horizons where the Taylor expansion's convergence radius is exceeded, Asgard supports *convergence tiling*: the time interval is partitioned into tiles, and the stream calculus expansion is applied within each tile, using the endpoint of one tile as the initial condition for the next.

5.4 Differentiability

Because all execution paths compile to JAX, circuits are fully differentiable regardless of which scheme is used. For circuits with feedback (trace), gradients propagate through fixed-point iterations via the implicit function theorem (§6 discusses this in detail). This enables gradient-based parameter identification and end-to-end optimisation of hybrid models.

6 Hybrid Systems and End-to-End Optimisation

The typed circuit representation opens several capabilities beyond simulating known equations.

6.1 Black-box components

Not every part of a dynamical system has a known analytic form. A chemical reaction rate, a turbulence closure, or a control policy may be best represented by a learned function—a neural network, a lookup table, or an arbitrary callable. Asgard supports this via the `black_box` atomic: a typed slot in the circuit that applies an external function to its input, with gradients flowing through it automatically via JAX autodiff.

A circuit might combine known physics (e.g., conservation laws expressed as `register`, `add`, `scalar`) with a learned correction term (`black_box`). The explicit structure captures what is known; the black-box component captures what is not. Crucially, the overall circuit remains typed and composable—the black box participates in the same categorical algebra as any other atomic. This bridges the gap between fully explicit models (interpretable but inflexible) and fully implicit models (flexible but opaque).

6.2 End-to-end optimisation through feedback

Because all execution paths compile to JAX, circuits are fully differentiable—including through `trace` (feedback loops). This is nontrivial: a traced circuit defines its output as a fixed point, so naive backpropagation through the iteration would require unrolling. Instead, Asgard applies the implicit function theorem: at the converged fixed point, the gradient is computed analytically via $\partial x^*/\partial\theta = -(I - \partial F/\partial x)^{-1}\, \partial F/\partial\theta$, implemented as a `jax.custom_vjp`.

This enables end-to-end optimisation of any parameter in the circuit—including parameters inside feedback loops, inside black-box components, or both. Given observed data, one can:

- Fit parameters of a known model structure (inverse problems).
- Train a neural network embedded within an otherwise explicit circuit (hybrid physics-ML).
- Jointly optimise structural parameters and learned components.

The optimiser sees the entire circuit as a differentiable function from parameters to predictions, regardless of internal complexity. Known physics provides inductive bias; learned components fill in the gaps—**explicit where possible, learned where necessary.**

6.3 Coupled systems

Many real-world systems consist of multiple interacting equations—predator-prey dynamics, multi-asset financial models, chemical reaction networks. Asgard supports coupled systems where each equation compiles to its own circuit, and the coupling structure is specified separately.

For stochastic systems, coupling includes correlated Brownian motions (specified as a scalar ρ for two equations or a full correlation matrix for n equations, factored via Cholesky decomposition). The executor determines the dependency structure automatically: unidirectional dependencies are resolved by topological ordering; bidirectional (circular) dependencies use simultaneous Picard iteration, where all equations are solved jointly in coefficient space.

Jump-diffusion processes—where the state can change discontinuously—are supported via the `jump_register` atomic, which encodes a Poisson integral $\int \cdot\, dN$. This enables models with both continuous diffusion and discrete jumps (e.g., Merton jump-diffusion, Kou double-exponential, Bates stochastic volatility with jumps), including coupled systems where some equations have jumps and others do not.

6.4 Circuit structure search

The rewrite system described in §3 is not only useful for simplification—it defines a *search space* over circuit topologies. Given a fitness function (e.g., mean squared error against observed data, penalized by circuit complexity), one can search for better circuit structures by applying rewrite rules and evaluating the result. Asgard supports greedy search, beam search, and genetic algorithms over this space, where each candidate is a valid, typed circuit and each move is a semantics-preserving (or approximately preserving) rewrite.

This turns the rewrite system into a neighbourhood operator for combinatorial optimisation over the space of dynamical models. Asgard provides the rewrite engine and search primitives; higher-level search orchestration (e.g., LLM-guided structure discovery) is handled by companion systems built on top of Asgard's circuit algebra.

7 From Equation to Circuit to Result

We trace three examples through the full pipeline—equation, compiled circuit, and execution under different semantics—to illustrate the syntax-semantics separation concretely.

7.1 The exponential: a deterministic ODE

Equation. The ODE $f' = f$ with $f(0) = 1$ is written as `f = f0 + int(f, x)`.

Circuit. The compiler produces (as in Eq. (1)):

$$\texttt{trace}\big(\texttt{composition}(\texttt{monoidal}(\texttt{var}(f_0), \texttt{composition}(\texttt{id}, \texttt{register}(x))), \texttt{add})\big)$$

The `trace` captures the recursive dependence of f on itself. Input degree 1 (the initial condition), output degree 1 (the solution).

Semantics. Under **RealCalculus**, `register` is formal power series integration. Picard iteration yields coefficients $c_n = 1/n!$—the Taylor expansion of e^x. Under **DiscreteCalculus**, the same circuit computes a discrete recurrence. The structure is unchanged; the meaning depends on the calculus.

7.2 Geometric Brownian motion: a stochastic process

Equation. The SDE $dS = \mu S\, dt + \sigma S\, dW$ is written as `Y = sde($mu * X, $sigma * X, X)`.

Circuit. Compilation produces a circuit containing `register` (drift integral $\int \mu S\, dt$) and `stochastic_register` (diffusion integral $\int \sigma S\, dW$), wired together with `trace` for the feedback.

Semantics. The same circuit admits multiple execution schemes. Under **stream calculus**, deterministic coefficients are computed once via Picard iteration, then combined with sampled iterated integrals across Monte Carlo paths. Under **Milstein**, the circuit is evaluated as a time-stepping integrator at strong order 1.0, using JAX autodiff for the diffusion derivative. Under **Euler–Maruyama**, a simpler strong-order-0.5 integrator is used. Both Itô and Stratonovich interpretations are available—the Stratonovich variant uses a Heun predictor-corrector. In all cases, the circuit is identical; only the execution semantics differ.

7.3 Fibonacci: a discrete recurrence

Equation. The recurrence $f_n = f_{n-1} + f_{n-2}$ with $f_0 = 0, f_1 = 1$.

Circuit. The circuit uses `trace` for feedback, `register` for the unit delay, and `multiplication` for the Cauchy product (convolution) with the recurrence coefficients $\sigma = (1, 1)$:

$$\texttt{trace}\Big(\texttt{composition}\big(\texttt{monoidal}(\texttt{param}(\sigma), \texttt{composition}(\texttt{id}, \texttt{register}(t))), \texttt{multiplication}\big)\Big)$$

Semantics. Under **DiscreteCalculus**, `register` is a unit delay and `multiplication` is sequence convolution. The circuit produces $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ Note the structural parallel with the exponential ODE: both use `trace` and `register`, but the calculus determines whether the result is a continuous function or a discrete sequence.

8 Related Work

Implicit dynamics learning. Neural ODEs [4] and PINNs [12] learn dynamics as black-box functions. Universal Differential Equations [11] embed neural networks within classical ODE solvers, and latent ODE models [13] extend the approach to irregular time series. These methods are flexible but fundamentally implicit.

Explicit structure discovery. SINDy [3] uses sparse regression over a predefined function library. Symbolic regression [15] searches via genetic programming. AI Feynman [16] combines neural networks with symbolic search. These recover explicit structure but are limited to narrow functional classes.

Equation-based modeling. Modelica [5] is an equation-based modeling language for multi-domain physical systems. It shares Asgard's philosophy of explicit structure but does not provide a typed compositional algebra, interchangeable semantics, or integration with learned structure search.

Symbolic-numeric systems. ModelingToolkit.jl [9] compiles equations to an intermediate representation with symbolic simplification and multiple solver backends, and is perhaps the closest existing system to Asgard in scope. However, it does not provide a categorical algebra for circuit equivalence, interchangeable calculi (real/stochastic/discrete) over a single representation, or integration with learned components. Diffrax [7] provides JAX-based differentiable ODE/SDE solvers with similar differentiability guarantees, but operates at the solver level rather than providing a typed compositional language for equation structure.

Stream calculus. Rutten's coinductive stream calculus [14] provides the mathematical foundation for Asgard's coefficient-by-coefficient ODE solving. We extend this to a computational framework with typed circuits, categorical composition, and JAX-based execution.

Categorical models of computation. Traced monoidal categories as models of feedback and recursion were introduced by Joyal, Street, and Verity [6]. Martin, Mathiesen, and Oliva [10] first showed that traced symmetric monoidal categories provide a sound and complete framework for Hoare-style reasoning about programs with loops and state; the full theory appears in [1]. Asgard builds on this line of work—developed by the present author and collaborators—using the categorical structure not only as a theoretical framework but as a practical type system and rewrite engine for dynamical systems.

9 Limitations and Status

Asgard's stream calculus approach has a finite convergence radius determined by the singularity structure of the solution; for long time horizons, convergence tiling (§5) mitigates this but does not eliminate it. Partial differential equations are not currently supported—the framework handles systems of ODEs and SDEs but not spatial discretisation. The fixed-point iteration used by `trace` involves Python control flow that prevents full JIT compilation of the Picard loop, though individual iterations compile to efficient JAX kernels. Non-smooth terms (e.g., $|x|$) may not converge at the expected rate in the Taylor expansion.

Asgard is implemented in Python/JAX and is under active development. This note describes the architecture and design; systematic benchmarks comparing accuracy, convergence, and performance against established solvers are forthcoming.

10 Conclusion

Asgard treats the mathematical structure of dynamical systems as a first-class computational object. By separating syntax (equations) from semantics (calculi) with a typed intermediate representation (circuits), it preserves the properties that make explicit models useful—interpretability, composability, verifiability—while supporting automatic differentiation, GPU acceleration, and stochastic simulation.

The bet is straightforward: if you can recover the structure, you get understanding for free. Implicit models give you predictions. Explicit structure gives you predictions *and* the ability to reason about why.

References

[1] Rob Arthan, Ursula Martin, Erik A. Mathiesen, and Paulo Oliva. A general framework for sound and complete floyd–hoare logics. *ACM Transactions on Computational Logic*, 11(1):7:1–7:31, 2009.

[2] Edwin S. Bainbridge. *Feedback and Generalized Logic*. PhD thesis, University of Michigan, 1976.

[3] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[4] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K. Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018. Best Paper Award.

[5] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley-IEEE Press, 2014.

[6] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

[7] Patrick Kidger. *On Neural Differential Equations*. PhD thesis, University of Oxford, 2022. Introduces Diffrax, a JAX-based differential equation solver library.

[8] Peter E. Kloeden and Eckhard Platen. *Numerical Solution of Stochastic Differential Equations*, volume 23 of *Stochastic Modelling and Applied Probability*. Springer, 1992.

[9] Yingbo Ma, Shashi Gowda, Ranjan Anantharaman, Chris Laughman, Viral Shah, and Chris Rackauckas. ModelingToolkit: A composable graph transformation system for equation-based modeling. *arXiv preprint arXiv:2103.05244*, 2021.

[10] Ursula Martin, Erik A. Mathiesen, and Paulo Oliva. Hoare logic in the abstract. In Zoltán Ésik, editor, *Computer Science Logic (CSL 2006)*, volume 4207 of *Lecture Notes in Computer Science*. Springer, 2006.

[11] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.

[12] Maziar Raissi, Paris Perdikaris, and George E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[13] Yulia Rubanova, Ricky T. Q. Chen, and David K. Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5321–5331, 2019.

[14] Jan J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.

[15] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

[16] Silviu-Marian Udrescu and Max Tegmark. AI Feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.