Lagra: A differentiable physics kernel for verifiable domain operations

Pim de Witte · *Lagra Project* · pim@medal.tv · May 19, 2026

Abstract

We argue that the Lagrangian, taken as a typed, additive object rather than as analytical shorthand, is the right framework for a single computational engine that spans molecular dynamics, rigid bodies, fluids, biomolecular folding, and high-energy physics. Concretely, we model the Lagrangian of a scene as a *graph* whose vertices are degrees of freedom and whose (hyper)edges are interaction terms drawn from a closed, dimensionally typed sum type. Time evolution is then a single *action kernel* — one loop that walks the graph, accumulates contributions to the action, and hands a generalized-force vector to a solver expressed as a program over the same graph. Conservation laws and other invariants enter as first-class *verifiers*: declarations that the kernel must satisfy, checked at every step by an independent checker in the style of a small-core proof assistant. We give the construction in enough detail to be reproduced, sketch why one kernel suffices across domains that historically demanded separate engines, discuss the position of this framework relative to existing differentiable simulators and to recent agentic AI workbenches for mathematics, and enumerate the obligations the design imposes on a runtime that aims to be useful as a verifiable execution environment for autonomous physics research.

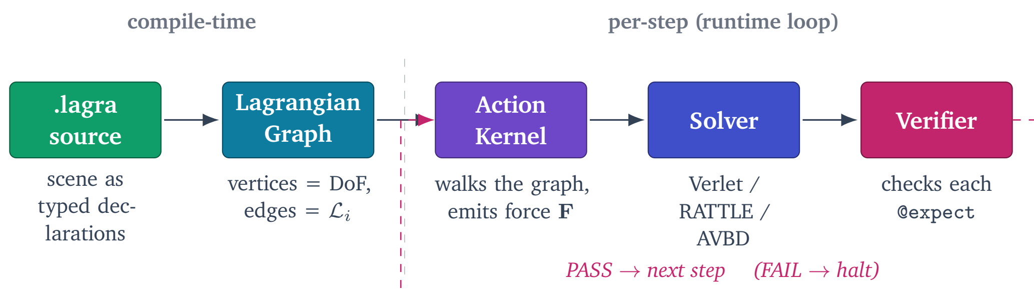


Figure 1: **The Lagrangian-graph pipeline in one figure.** A scene written in `.lagra` is compiled, once, into a **Lagrangian Graph**: a static typed data structure whose vertices are degrees of freedom (particles, rigid bodies, torsions) and whose edges are additive interaction terms drawn from a closed vocabulary (`HarmonicBond`, `Coulomb`, `LennardJones`, ...). At every step the **Action Kernel** — implemented in `lagra-core` as a struct called `ActionGraph`, separate from the `LagrangianGraph` — walks the active edges once and emits a generalized-force vector. The **Solver** (itself a program over the same graph: Verlet, RATTLE, AVBD, ...) advances the state. The **Verifier** checks each declared invariant (`@expect energy_conservation`, `@expect charge_conservation`, ...) against a stated tolerance. A PASS feeds the next step; a FAIL halts the run in CI, or flags the offending step in the editor (where the renderer can then draw it). The same kernel runs the two-bead oscillator (Sec. 5), the thousand-block AVBD stack, the dam-break fluid scene, and the DNA-helix demo of (de Witte, 2026); only the active edges in the Lagrangian Graph differ between them.

1. Introduction

Lean (de Moura and Ullrich, 2021; The mathlib Community, 2020) put mathematics into a *verifiable* domain, accelerating the field by making proofs machine-checkable and therefore, in part,

machine-discoverable (Polu and Sutskever, 2020; Zheng et al., 2026). We aim for analogous progress in physics: express physical theories as Lagrangians and validate them against observation under a runtime whose conformance to the declared physics is itself machine-checked. The Lagrangian formulation is uniquely suited to this role: every interaction in the Standard Model is an additive contribution to one Lagrangian density, every conservation law is — by Noether's theorem (Noether, 1918) — a continuous symmetry of that Lagrangian, and the same formalism scales from femtometre molecular dynamics to relativistic charged-particle motion in astrophysical fields (Goldstein et al., 2002).

Conventional physics engines do not behave as if they had noticed. Rigid-body engines (Unreal, Genesis, MuJoCo), classical many-body codes (Rosetta), and RNA tools (ViennaRNA) each ship as a separate binary with a separate input format and a separate idea of what "conservation" means at the API boundary. Each cheats somewhere to make the arithmetic tractable: cars are wheels-on-spheres, fluids in a rigid-body engine are bolted on, electrostatics in an RNA folder means rewriting the engine. The Lagrangian all of them are, at root, evaluating numerically is nowhere in the type system. This is how the community has ended up with several dozen physics engines and no unifying interface.

We propose a different organization. A scene is written in a small DSL (`.lagra`) as a typed sum of Lagrangian terms; the engine compiles it to a graph and walks the graph each step with a single loop. Interaction terms and solvers are swappable at runtime, with no domain-specific branches in the inner loop. Because each contribution is a small piece of code with a known signature, the engine produces exact differentials across long trajectories — useful for reinforcement learning, for discovery loops that propose modifications to the Lagrangian, and for learning dynamics directly from environment gradients. The construction is a triple:

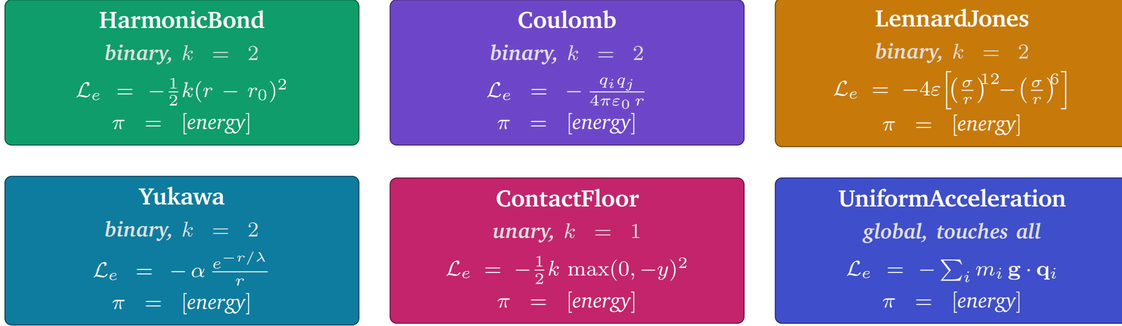
1. a *Lagrangian graph*, a typed graph whose vertices are degrees of freedom and whose edges are terms drawn from a closed, dimensionally checked sum type of physical interactions;
2. an *action kernel*, a single evaluator that walks the active edges and accumulates each term's contribution to a generalized force, with no special-casing on physics domain; and
3. a layer of *verifiers*, propositions about the run (*e.g.* $|\Delta H|/H_0 < 10^{-5}$) discharged at every step by an independent checker in the style of a proof-assistant kernel.

Figure 1 (above) is the whole pipeline in one picture, with the two graphs of the reference implementation kept distinct: a static **Lagrangian Graph** (what interaction terms exist) and a per-step **Action Kernel** (the loop that walks it, implemented in `lagra-core` as a struct called `ActionGraph`). The same engine has been exercised on classical many-body mechanics and on Boris-style integrators for charged particles in electromagnetic fields (Boris, 1970; Vay, 2008; Higuera and Cary, 2017) — the "one engine, many scales" position the Lagrangian formulation invites. The rest of the paper formalizes each of the three pieces (Section 2–4), works the simplest non-trivial example end-to-end (Section 5), names the five design commitments the construction rests on (Section 6), argues why one kernel suffices (Section 7), and places the framework relative to differentiable simulators, proof assistants, and agentic AI for mathematics (Section 8–9).

2. The Lagrangian graph

We now make the framework precise. Let $\mathbf{q} \in M$ be the configuration of the system, where M is a smooth manifold of generalized coordinates (positions of particles in $\mathbb{R}^3$, Euler angles or quaternions for rigid bodies, RNA torsion angles, etc.). Let $\dot{\mathbf{q}} \in T_{\mathbf{q}}M$ be the generalized velocity.

$$\text{Kind} \;=\; \texttt{HarmonicBond} \mid \texttt{Coulomb} \mid \texttt{LennardJones} \mid \texttt{Yukawa} \mid \texttt{ContactFloor} \mid \texttt{UniformAcceleration} \mid \ldots$$



Every variant carries a fixed arity $k(e)$, a typed parameter set, and a dimensional signature π that the compiler checks before any numerical work runs. Adding a new physics term is adding one case to this enumeration.

Figure 2: **The vocabulary** Kind **of** `LagrangianNode` **variants.** Six representative cases of the closed sum type the engine ships with (the full enumeration is roughly two dozen, including `NuclearYukawa`, `QcdCoulomb`, `MorseBond`, `FeneBond`, `HydrogenBond`, `GravityWell`, ...). Each variant declares its arity, parameters, and units signature once; every legal edge in any scene is an instance of one of these. The chip colours match Figs. 3 and 5 so the same variant is the same colour throughout the paper.

Definition 1 (Lagrangian graph)**.** A *Lagrangian graph* is a tuple $\mathcal{G} = (V, E, \tau, \pi)$ where:

- V is a finite set of *degree-of-freedom vertices*; each $v \in V$ carries a type signature $\tau(v)$ (*e.g.* a particle in $\mathbb{R}^3$, a rigid body in $SE(3)$, a torsion angle in S^1) and a slice of the state vector $\mathbf{q}$;
- E is a finite set of *Lagrangian (hyper)edges*; each $e \in E$ has a fixed arity $k(e)$, a tuple of incident vertices $\partial e \in V^{k(e)}$, and a *kind* drawn from a closed sum type Kind (`Coulomb`, `LennardJones`, `HarmonicBond`, `ContactFloor`, `UniformAcceleration`, `QcdCoulomb`, ...);
- π is a dimensional typing rule that assigns each edge kind a units signature, so that $\pi(\mathcal{L}_e) = \pi(\mathcal{L}_{e'})$ for all $e, e' \in E$.

Property 1 (Additivity of $\mathcal{L}$)**.** The Lagrangian of the scene is the sum of edge contributions:

$$\mathcal{L}(\mathbf{q}, \dot{\mathbf{q}}, t) \;=\; \sum_{e \in E} \mathcal{L}_e(\mathbf{q}|_{\partial e}, \dot{\mathbf{q}}|_{\partial e}, t), \tag{1}$$

where $\mathbf{q}|_{\partial e}$ is the projection of $\mathbf{q}$ onto the slices indexed by ∂e. The action of the scene over $[t_0, t_1]$ is $\mathcal{S} = \int_{t_0}^{t_1} \mathcal{L}\, dt$.

Figure 2 lays out the closed sum type Kind itself: six representative variants of `LagrangianNode` (the implementation carries roughly two dozen) with the arity, the Lagrangian expression, and the units signature for each. Every legal edge in any scene is one of these variants. Figure 3 then makes a concrete instance: four particles, six edges of four distinct variants, plus two global terms. The Lagrangian is just the sum of the labelled contributions; the kernel of Section 3 is the loop that walks them.

Three remarks make this less abstract than it reads. First, additivity is not a modelling assumption — it is just the universal property of Lagrangians as physicists already use them. Writing the Lagrangian density of the Standard Model as a sum of gauge, fermion, and Higgs

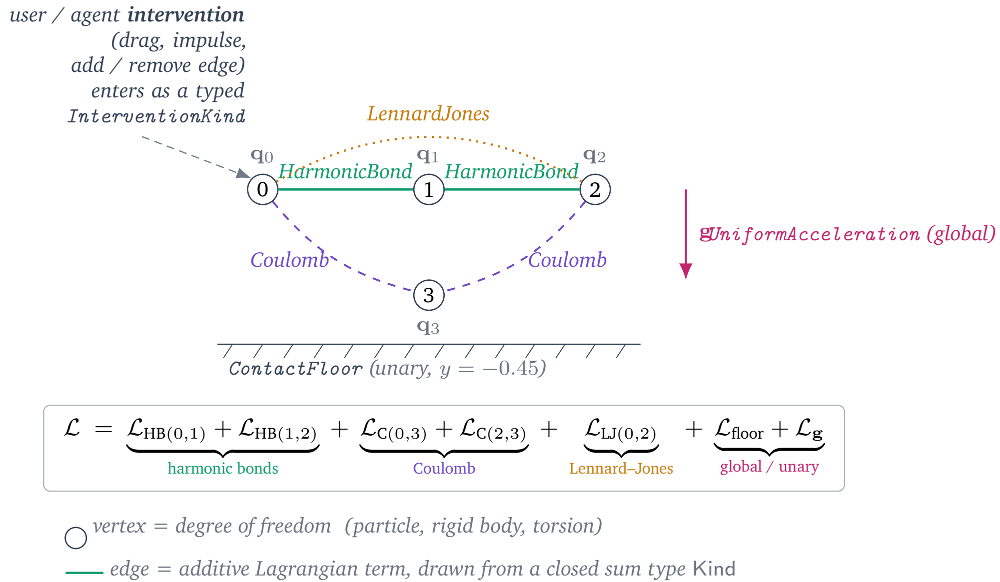


Figure 3: **Anatomy of a LagrangianGraph.** Four particles ($q_0, \ldots, q_3$) and six interaction terms drawn from four different variants of the closed sum type Kind: two binary `HarmonicBond`s along the chain, two binary `Coulomb` edges to the lower particle, one binary `LennardJones` edge across, and two global / unary terms (gravity and a ground-contact floor) that touch every particle without being attached to a specific pair. The Lagrangian is the labelled sum at the bottom. The kernel (Section 3) iterates over these edges once per step and asks each for its contribution to the generalized force; nothing in the kernel knows or branches on which variant a given edge is.

terms is the same move. Second, the closed sum type Kind is not a barrier: extending the engine to a new physics domain means adding a variant, which is a localized, type-checked change. In the reference implementation the variants are the same nouns one would use in a textbook, declared as a single Rust enumeration:

```
pub enum LagrangianNode {
    HarmonicBond { i: usize, j: usize, k: f64, r0: f64, damping: f64 },
    Coulomb      { coupling: f64 },
    LennardJones{ epsilon: f64, sigma: f64, cutoff: f64 },
    Yukawa       { coupling: f64, range: f64, cutoff_factor: f64 },
    ContactFloor{ y: f64, k: f64, damping: f64, friction: f64 },
    UniformAcceleration { g: [f64; 3] },
    // ... ~24 variants in the shipped enumeration
}
```

Third, the typing rule π is what makes the framework *verifiable*: an attempt to add a `HarmonicBond` with k in units of an electric charge is rejected at parse time, not at runtime when the energy diverges.

Two graphs, not one. It is important not to conflate $\mathcal{G}$ — the *Lagrangian graph* defined above — with the per-step force evaluation that the runtime actually walks. We follow the distinction made in the reference implementation between a *Lagrangian graph* (which energy terms exist, static per scene; in the source, the type `LagrangianGraph`) and an *action kernel* (compute total force given the current state, dynamic per step; in the source, a struct `ActionGraph` whose role is to traverse a `LagrangianGraph` once per step). The Lagrangian graph defines the rules; the action kernel traverses them. Traversal strategies (serial loop, cell list, Barnes–Hut, GPU kernel launch) are implementation details of the kernel and are not visible to the physics spec-

ification. This separation is what lets the engine ship a single source of truth for the physics and several traversal back-ends. The two roles are unfortunately both natural to call "the action graph" in English; we reserve *action graph* for the collective object that is the typed pair (*Lagrangian graph, action kernel*), and use the two more precise names whenever the distinction matters.

3. The action kernel

The contract of a single timestep is the cleanest thing in the framework, and it is worth showing as a data-flow diagram before any code. Figure 4 draws it: the two inputs ($\mathcal{G}$ and the current state) are typed, the **kernel** is a single function from those inputs to a generalized-force vector $\mathbf{F}$ (and, optionally, the scalar Lagrangian value $\mathcal{L}$), and the **solver** is a second function from $\mathbf{F}$ and the current state to the next state. The kernel is side-effect-free except for writing $\mathbf{F}$; the solver is side-effect-free except for writing the next state; everything that matters about one timestep is on the arrows.

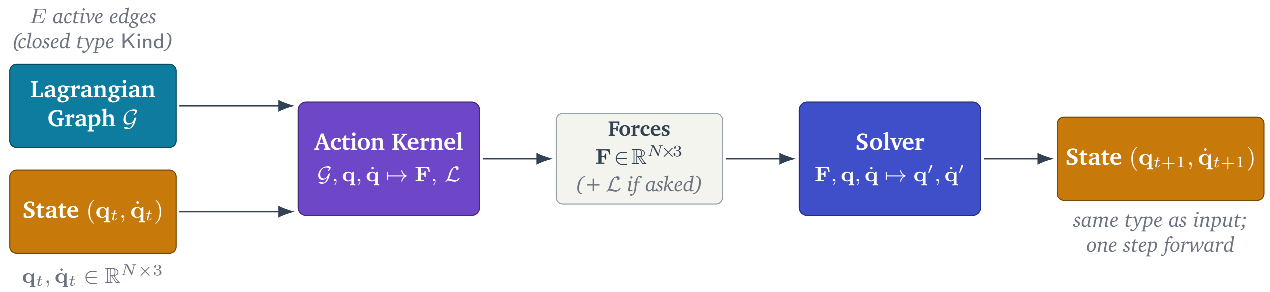


Figure 4: **One timestep, as a data-flow diagram.** Inputs (left) are the static **Lagrangian Graph** $\mathcal{G}$ and the current per-particle **State** $(\mathbf{q}_t, \dot{\mathbf{q}}_t)$. The **Action Kernel** is a single function that walks $\mathcal{G}$'s active edges and emits the generalized-force vector $\mathbf{F} \in \mathbb{R}^{N\times 3}$; if a verifier asked for it, the kernel also returns the scalar Lagrangian value $\mathcal{L}$ at the current state. The **Solver** is a second function that consumes $\mathbf{F}$ together with the current state and produces the next state. Nothing else flows between the two; both functions are side-effect-free with respect to anything other than their declared output. The same diagram is repeated K times by the action-descent solver of Eq. (2), with the gradient walked over a window of $K+1$ states instead of a single one.

The kernel in pseudocode — the box labelled *Action Kernel* in Fig. 4 — is essentially one loop:

```
fn kernel(graph: &LagrangianGraph,
          state: &State,
          forces: &mut Forces) {
    forces.zero();
    for node in graph.active_nodes() {
        node.accumulate(state, forces);
    }
}
```

Figure 5 zooms into the *Action Kernel* box of the data-flow diagram (Fig. 4) and exposes its internals: the heterogeneity that distinguishes a Lennard-Jones term from a harmonic bond lives in each variant's `accumulate` method, not in the loop. The loop iterates over whichever edges the scene declared active and dispatches to each one in turn; the variant implementations sit below as a menu of small, independent functions that all share the same signature.

The contract is deliberately narrow. `accumulate` reads `state` and adds to `forces`; it does not allocate, it does not branch on simulation-wide configuration, and it is the only piece of

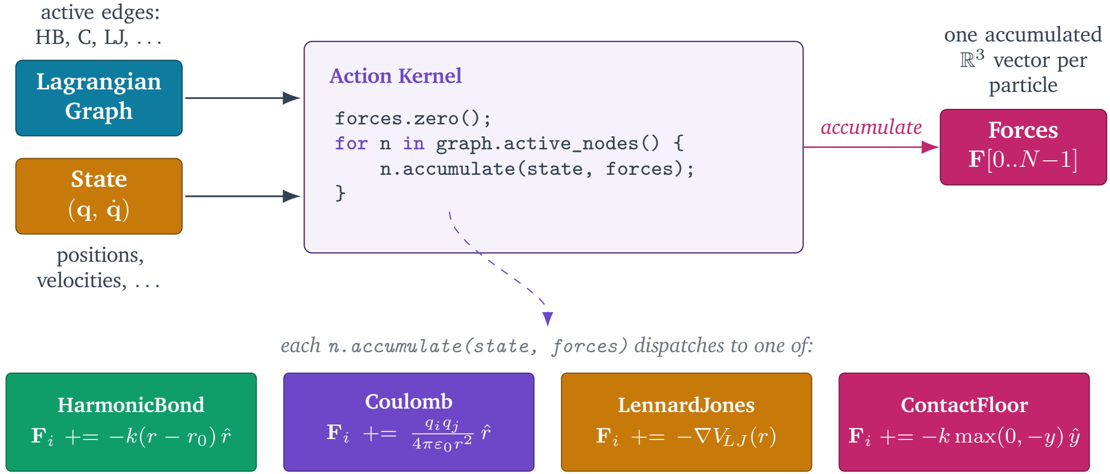


Figure 5: **The action kernel, exploded.** The Action Kernel (top centre) is the single loop that turns the Lagrangian Graph's active edges plus the per-particle State $(\mathbf{q}, \dot{\mathbf{q}})$ into a shared Forces buffer. The heterogeneity lives in the `accumulate` method of each variant (bottom menu: four examples). Every `accumulate` has the same signature `(state, forces)`; the kernel itself never branches on which variant it is calling. Adding a new physics term is adding a new `accumulate`, not editing the loop.

code allowed to know what its own term means physically. The kernel itself is term-agnostic: it iterates and dispatches. A scene with one harmonic bond walks one node; a scene with a million Lennard-Jones pairs walks a million (typically with the help of a cell-list traversal that the planner picks). The cost scales with the energy density of the system the user described, not with the engine's full feature set.

Why one loop suffices. The closure of Kind under additivity is the structural reason that one loop is enough. Each $\mathcal{L}_e$ is itself a small piece of code with a known signature, so the kernel produces $\mathbf{F}_e = -\nabla_{\mathbf{q}|_{\partial e}} \mathcal{L}_e$ analytically, and the sum is what Property 1 guarantees is the right total. The kernel never needs to know whether a term is "gravitational" or "electrostatic"; it asks the term for its contribution to $\mathbf{F}$ and adds it to the buffer.

The solver as a program over $\mathcal{G}$. The kernel produces forces; a *solver* consumes forces and produces the next state. We treat the solver in exactly the same way as the Lagrangian: as an expression over the same graph, drawn from a small closed vocabulary. Velocity Verlet is the program $\dot{\mathbf{q}} \mathrel{+}= \frac{1}{2}\Delta t\,\mathbf{F}(\mathbf{q})$; $\mathbf{q} \mathrel{+}= \Delta t\,\dot{\mathbf{q}}$; $\mathbf{F} \leftarrow \mathrm{kernel}(\mathcal{G}, \mathbf{q}, \dot{\mathbf{q}})$; $\dot{\mathbf{q}} \mathrel{+}= \frac{1}{2}\Delta t\,\mathbf{F}$. Symplectic Euler, RATTLE, projective dynamics, position-based dynamics (Müller et al., 2007), and the Vertex-Block-Descent family used for contact-rich rigid-body and deformable scenes (Chen et al., 2024, 2025) are different programs over the same operations.

Action descent: the general solver. The most useful solver in the framework, and the one used by default in the demos of (de Witte, 2026), is not a fixed-stencil integrator at all but a *discrete-action minimizer* (Strang et al., 2023), called *action descent* in the source. Given a trajectory window $\mathbf{q}_{0:K}$ of $K+1$ states at spacing Δt, the discrete action is

$$\mathcal{S}_d[\mathbf{q}_{0:K}] = \sum_{k=0}^{K-1} \Delta t\, \mathcal{L}\left(\frac{\mathbf{q}_{k+1}+\mathbf{q}_k}{2}, \frac{\mathbf{q}_{k+1}-\mathbf{q}_k}{\Delta t}\right), \tag{2}$$

and the discrete Euler–Lagrange equations are recovered by setting $\partial \mathcal{S}_d / \partial \mathbf{q}_k = 0$ for each interior k. The action-descent solver computes this gradient (using the kernel of Section 3 to assemble the per-edge contributions) and walks the trajectory window down the gradient with adjustable step size and iteration count, both of which are scene parameters. In our implementation, defaults of order two thousand inner iterations and a sixteen-state window suffice for the demos; harder scenes raise both numbers. The construction is exactly the discrete-mechanics / variational-integrator setting of Marsden and West (2001), lifted to a runtime in which $\mathcal{L}$ is itself the user-facing object. Two properties matter for us. First, the solver is a single program over $\mathcal{G}$: it never inspects which variants are present, only the gradient of $\mathcal{S}_d$ that the kernel returns. Second, because the objective is the action and not the equations of motion directly, constraints and dissipation enter the same way: through additional edge variants whose Lagrangian contributions are added to $\mathcal{L}$. In the reference implementation, articulated bodies under projected contact are solved this way (the internal solver mode is named `CpuArticulationActionDescent`), and rejected steps re-enter the loop without any per-domain rewrite.

4. Verifiers

If the kernel is the engine of the framework, the verifier layer is its ethics. A scene declares, alongside its entities and interactions, a list of propositions that any acceptable run must satisfy. In the reference DSL these are `@expect` clauses (`@expect energy_conservation`, `@expect momentum_conservation`, `@expect charge_conservation`, `@expect dimensional_consistency`, `@expect dna_helix_stable`, *etc.*). At every step the runtime emits a numeric witness for each proposition; an independent checker compares the witness to a stated tolerance and returns PASS or FAIL. A FAIL halts the run — or, in interactive mode, flags the offending step in the editor; in CI mode it fails the build.

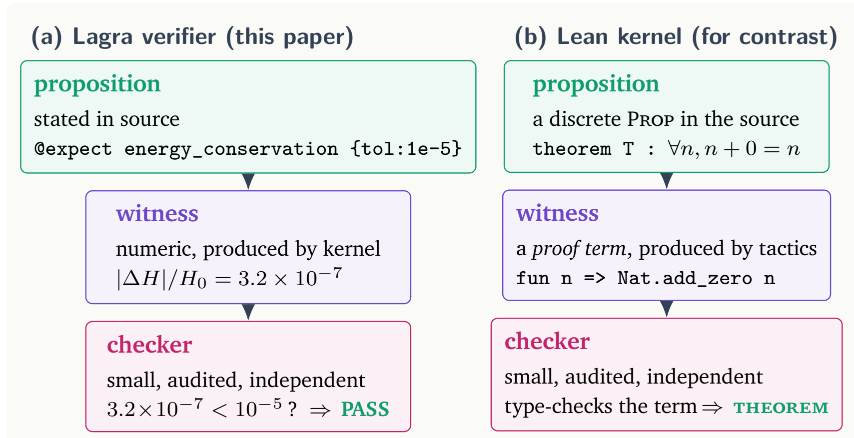
The *structural* architecture is the architecture of a proof assistant (de Moura and Ullrich, 2021). There are three pieces, with sharp interfaces between them:

1. a *proposition* (here: an invariant like $|\Delta H|/H_0 < \epsilon$), stated in source;
2. a *witness* — here a number produced by the kernel/solver, there a proof term produced by tactics — that is supposed to discharge the proposition;
3. a *checker* — a small, audited piece of code — that accepts or rejects the witness.

The checker is independent of the part of the system it is checking. The kernel is allowed to be fast, vectorized, GPU-resident, and performance-tuned; the checker is allowed to be slow, single-threaded, and obvious. This separation is what makes "the simulator says energy was conserved" meaningful — the entity that did the math is not the entity that signed off on it.

Figure 6 makes the three-piece architecture explicit and contrasts it side-by-side with the corresponding pieces in a proof-assistant kernel; the comparison is what motivates the disanalogy paragraph below.

The disanalogy with Lean is real and worth stating. A Lean checker discharges a deductive obligation: given a proof term, it returns a yes/no decision that, modulo bugs in a small audited kernel, is a mathematical theorem. The Lagra verifier discharges a *numerical* obligation: given a floating-point witness and a user-stated tolerance, it returns a yes/no decision that is contingent on the choice of tolerance, on the stability of the integrator, on finite precision, and on the run length. The two are not the same trust event. We borrow the proof-assistant *discipline* — one small audited checker, an independent witness, a proposition stated in source — not its



Same structural shape; different trust event. The Lagra checker discharges a numerical obligation contingent on tolerance, integrator stability, and finite precision; the Lean checker discharges a deductive obligation that, modulo bugs in the kernel, is a theorem. We borrow the discipline, not the guarantees.

Figure 6: **Verifier triangle, side by side with its Lean analogue.** A scene's `@expect` clauses (a) play the role of Lean propositions (b); the kernel emits a numeric witness where Lean tactics emit a proof term; an independent small checker decides PASS in both cases. The next paragraph spells out why "PASS" nevertheless means two different things.

guarantees. A PASS from a Lagra verifier means "no declared invariant has drifted outside the declared tolerance during this run"; it does not, and is not meant to, mean "a theorem has been proven." Throughout the paper, we use "verifier" in this weaker numerical sense. Where we want the stronger sense (for example, in arguing that Noether-derived conservation laws hold by construction), we say so explicitly.

Dimensions. A particular case worth singling out is dimensional consistency. Because π assigns each $\mathcal{L}_e$ a units signature and Property 1 requires they agree, the scene compiler can reject a Lagrangian whose terms cannot be summed before any numerical work happens. We treat dimensional analysis as type-checking, in the same sense and with the same payoff that the ML community treats shape-checking as type-checking in tensor programs (Hattori et al., 2022). A simulator that has never been allowed to compute $\mathrm{N} + \mathrm{N} \cdot \mathrm{m}$ does not need to explain what it would have done.

Noether's theorem as a verifier-generator. Continuous symmetries of $\mathcal{L}$ generate conserved quantities, by Noether's theorem (Noether, 1918). In the framework, this is a constructive procedure: from a declared translational symmetry of an edge kind, the runtime can derive the linear-momentum invariant; from rotational symmetry, angular momentum; from time-translation, energy. Verifiers therefore are not a fixed list provided by the engine authors; they are generated, in part, by the same algebra that produces the equations of motion.

5. Worked example: the two-bead oscillator

To make the construction concrete, we run the simplest non-trivial scene end to end. The full source is below; everything that follows is generated from it.

```
scene oscillator {
    uses newtonian.classical_mechanics
```

```
    description "Two-bead harmonic oscillator."

    entity { type: ball_small, position: [-1.35, 0, 0], mass: 1.0 }
    entity { type: ball_small, position: [ 1.35, 0, 0], mass: 1.0 }

    bond harmonic [0, 1] { k: 2.0, r0: 2.0, damping: 0.04 }

    @expect momentum_conservation
    @expect energy_conservation { tol: 1e-5 }

    solver { integrator: "velocity_verlet", dt: 0.002,
             steps_per_frame: 12 }
}
```

The compiler turns this into three typed runtime objects:

- a `LagrangianGraph` with two vertices and one edge of kind `HarmonicBond`, parameters ($k = 2.0$, $r_0 = 2.0$, $\gamma = 0.04$);
- a `FormulaStepper` encoding velocity Verlet with $\Delta t = 2 \times 10^{-3}$ and twelve sub-steps per frame; and
- a `Verifier` bundle of two propositions: momentum conservation (no tolerance: must hold to floating-point round-off, because the kernel is translation-symmetric by construction) and energy conservation to relative tolerance 10^{-5}.

At each step the kernel walks one node, the solver applies the half-kick/drift/half-kick program, and the verifier records $|\Delta H|/H_0$. The Lagrangian is

$$\mathcal{L} = \tfrac{1}{2}m_0\|\dot{\mathbf{q}}_0\|^2 + \tfrac{1}{2}m_1\|\dot{\mathbf{q}}_1\|^2 - \tfrac{1}{2}k(\|\mathbf{q}_1 - \mathbf{q}_0\| - r_0)^2, \tag{3}$$

and the force on bead i is $\mathbf{F}_i = -k(\|\mathbf{q}_1 - \mathbf{q}_0\| - r_0)\,\hat{\mathbf{q}}_{ij}$. The point of writing this out is not the physics, which is textbook, but the chain of custody: every token in the scene source corresponds to a typed runtime object, the kernel sees nothing else, and the verifier can sign each step against the declared invariants.

5.1. The trajectory is consistent with the law it was simulated under

A Lagrangian is the mathematical object that defines a physical theory: every term in the Standard Model lives inside one, every classical mechanics problem can be written as one, and the equations of motion follow by applying the Euler–Lagrange operator. A `.lagra` scene declares one — each entity contributes a kinetic term to $\mathcal{L}$, each interaction term contributes a potential term, and the graph of Fig. 1 is what that declaration compiles to. Because evolving the graph forward in time amounts to integrating the Euler–Lagrange equations for the declared $\mathcal{L}$, the resulting trajectory is, up to integrator error, a solution of those equations. It is therefore consistent with the laws of physics that $\mathcal{L}$ encodes, and any observable those laws predict in closed form should be recoverable from the output by direct measurement. We call this property *verifiability-by-derivation*: the trajectory and the analytical theory predict each other, and their agreement is itself the validation.

For the oscillator of Eq. (3) this is sharp. The closed-form analysis of $\mathcal{L}$ gives one normal mode at angular frequency $\omega = \sqrt{k/m_{\text{red}}}$ with reduced mass $m_{\text{red}} = m_0 m_1/(m_0 + m_1) = 1/2$, hence $\omega = \sqrt{k/m_{\text{red}}} = \sqrt{4} = 2$ in the natural units of Eq. (3). The engine never sees this formula; it only evolves the graph. Yet the period measured directly from the bead separation $r(t) - r_0$ recovered from the simulation matches $T = 2\pi/\omega$ to the residual error of velocity Ver-

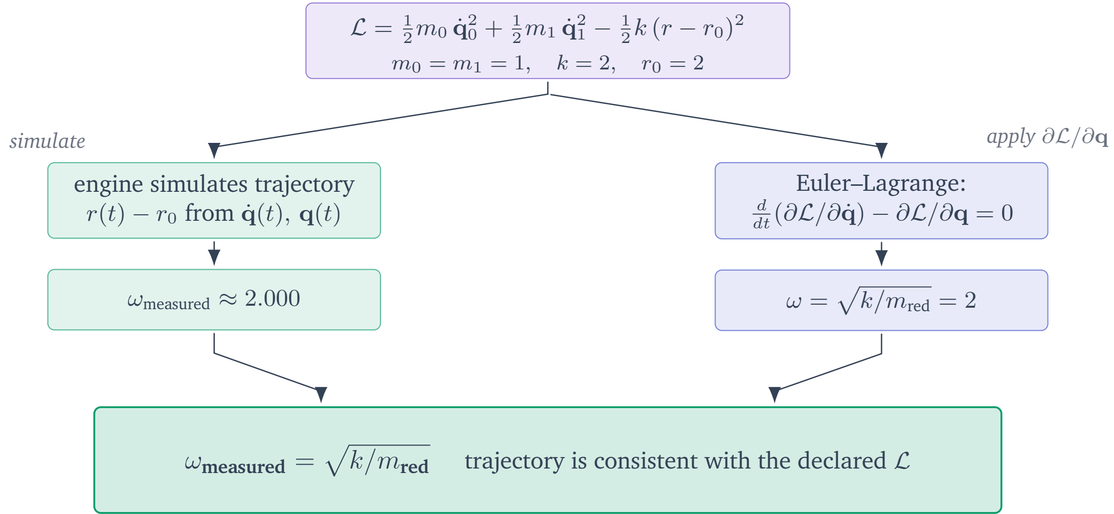


Figure 7: **Verifiability by derivation.** The same Lagrangian flows down two independent branches: the engine evolves the LagrangianGraph forward in time and the bead-separation period of the resulting trajectory yields ω_{measured}; classical analysis applies the Euler–Lagrange operator to $\mathcal{L}$ and reads off $\omega = \sqrt{k/m_{\text{red}}}$ in closed form. The two estimates agree to integrator error, and that agreement *is* the verification. No separate test asserts it; the trajectory itself is proof that the kernel solved the declared physics. The same construction recovers Kepler's third law, the Maxwell–Boltzmann distribution, and arbitrary Noether charges; a separate paper develops the analysis pipeline for the full Standard-Model coupling tree.

let at the chosen Δt. Fig. 7 makes the two-branch construction explicit. The same construction generalises: Kepler's third law is recoverable from a two-body gravitational scene, the Maxwell–Boltzmann velocity distribution from an equilibrated Lennard-Jones liquid, and any Noether charge from a continuous symmetry of $\mathcal{L}$. The full treatment of how these checks compose across coupled subsystems (mechanical, electromagnetic, nuclear) is the subject of a separate forthcoming paper; here we restrict to noting that the construction is well-defined exactly because $\mathcal{L}$ is the input.

What is gained by this discipline is what is gained by writing a small Lean proof for a textbook result: the practitioner has not learned new physics, but the next ten thousand variations of this scene now ride on a foundation that does not silently violate the textbook result. The same kernel and the same verifier code are what gates the thousand-block rigid-body stack, the dam-break fluid scene, the DNA helix, and the toy Standard-Model interactions used as live demos of the engine (de Witte, 2026).

6. Design principles, in retrospect

The construction the reader has now seen is the consequence of five commitments. Each is the sharp edge of a trade-off; naming them explicitly makes it easier to see what the framework is buying and what it is giving up.

One Lagrangian per scene, made out of typed parts. A scene's physics is a single mathematical object: the Lagrangian $\mathcal{L} = \sum_i \mathcal{L}_i$. Each $\mathcal{L}_i$ is drawn from a closed, finite vocabulary of interaction terms (Coulomb, Yukawa, Lennard-Jones, harmonic bond, FENE bond, Morse bond, contact-with-floor, uniform acceleration, nuclear Yukawa, QCD-color Coulomb, ...). The vocabulary is expressible at the type level: in our reference implementation the sum type is a Rust `enum LagrangianNode` with one variant per term and the dimensional content of each coupling

fixed at compile time. Adding a new term means adding a variant; it does not mean editing the kernel, the solver, or the verifier. The vocabulary is closed not because we think physics is closed but because the kernel must be able to enumerate what it is about to evaluate.

The graph is data, the kernel is a single loop. Once the Lagrangian is a sum of typed terms, the natural data structure is a graph: vertices are degrees of freedom and (hyper)edges are the terms that couple them. The evaluator is then a single loop over active edges that accumulates each edge's contribution to a shared force buffer. There is no top-level branch on physics domain. A scene with one harmonic bond and a scene with 10^6 Lennard-Jones interactions are executed by the same code path; what differs is which edges are in the graph and how the planner schedules the loop (serially, with a cell list, on the GPU). The kernel does not know it is doing "molecular dynamics" or "rigid bodies" — the user expressed the difference by picking different `LagrangianNode` variants.

Solvers are programs over the graph, not subclasses. Integration — velocity Verlet, symplectic Euler, RATTLE, projective dynamics, AVBD, action descent (Strang et al., 2023) — is the second characteristic-engine-shaped wart in conventional codes. We take the same position as for the Lagrangian: the integrator is itself an expression over the graph (a *half-kick, drift, evaluate, half-kick* program, in the symplectic case) rather than a method on a hardcoded class hierarchy. This is what lets us swap integrators without rewriting the engine, and to derive equations of motion automatically (via differentiation of $\mathcal{L}$) when the user did not supply them.

Verifiers are propositions; the checker is independent. A scene declares the invariants it expects (energy conservation, momentum conservation, dimensional consistency, charge conservation, positivity of the metric, . . .). At every step the runtime emits a proof obligation — a numeric witness — and an *independent* small checker decides PASS or FAIL with a stated tolerance. The structure is deliberately analogous to a proof-assistant kernel à la Lean (de Moura and Ullrich, 2021; The mathlib Community, 2020): the simulator proposes, the checker disposes. The checker's code is small, audited, and shared between continuous integration, the editor's debug panel, and the in-browser demos. This is the property that makes the simulator useful as the execution environment of an autonomous physics agent (Zheng et al., 2026; Polu and Sutskever, 2020): a run that the verifier rejects is not silently shipped as evidence.

Rendering and observation are pure functions of state. The kernel and the solver produce a sequence of states $\mathbf{q}_0, \mathbf{q}_1, \ldots$. Everything downstream — the CPU rasterizer that draws the demos in a web browser, the JSONL trace dumper used for quantitative analysis, the agent-facing API that returns observables like kinetic energy or end-to-end RNA distance — is a pure function of that sequence. Headless agent harnesses pay nothing for rendering they do not use; conversely, a viewer can attach to a long-running simulation without perturbing it. The simulator is a state-producing program; the GUI is a separate program that consumes the same protocol.

7. Why one kernel is enough

Three questions are worth answering directly.

7.1. What stops the closed sum type Kind from exploding?

Empirically, the variants required to cover the part of physics we have so far targeted form a short list. The reference implementation currently carries on the order of two dozen variants and these suffice for Newtonian many-body mechanics, rigid-body contact with friction (AVBD), constrained chains, Lennard-Jones fluids, electrostatics, Yukawa fifth-force studies, nuclear and

weak Yukawa terms, QCD-color Coulomb interactions, harmonic and Morse and FENE bonds for biopolymers, contact with a floor, and uniform external acceleration. The reason the list has not exploded across these domains is that they factor cleanly into additive, low-arity terms: "simulate a protein in water" is not a new variant, it is a particular incidence of the Lennard-Jones, Coulomb, hydrogen-bond, and harmonic-bond variants.

The honest qualification is that the closed vocabulary as it stands is *additive in pairwise + uniform-field terms*, and a number of physically important interactions do not factor this way. Examples include 4-body dihedral and improper terms (essential for protein backbones and aromatic ring stability), genuinely many-body bond-order potentials such as Tersoff, Stillinger-Weber, and the EAM family (needed for covalent solids and metals), polarizable force fields (needed for accurate water and ion chemistry), the full electroweak gauge sector, gravitational radiation reaction, and any framework with non-Markovian memory kernels. Each of these is an extension axis: a new edge arity ($k = 4$ for dihedrals; k depending on a coordination shell for bond-order potentials), or a new variant whose `accumulate` reads from a polarization field carried in `State`. We treat "can the engine express this?" as a type-system question with a known answer for each axis above, but do not claim the current enum is complete.

7.2. What about constraints?

Holonomic constraints (bond lengths, joint angles, rigid-body compositions) are handled by projection or by Lagrange multipliers in the solver, not by a special branch in the kernel. The kernel still sees only $\mathcal{L}_e$ terms. The constraint solver (RATTLE for SHAKE-style distance constraints, AVBD for primal-dual contact, projective dynamics for soft constraints) is just one more program over the graph, in the same sense that velocity Verlet is. The framework's job is to make this choice composable, not to force a single integrator on every scene.

7.3. Compile-time term elimination and coarse-graining.

Because the Lagrangian graph is data and is fully known before any inner loop runs, the compiler can prune. Variants that no scene references generate no code; pairwise terms whose cutoff radius exceeds the simulation box never enter the broad-phase; bonded exclusions are folded into the cell-list construction. The practitioner who wants electromagnetism in a biopolymer scene adds the `Coulomb` variant to the scene's active set and pays for the broad-phase that comes with it; the practitioner who wants a domino stack pays for neither electromagnetism nor the broad-phase. The same mechanism supports coarse-graining: an entity declared as a "residue" contributes one degree of freedom, not the dozen its constituent atoms would; the kernel is unaware of the rescaling because it sees only the edges the scene exposed. None of this prevents a separate scene, with the same engine, from re-introducing the finer-grained entities later in the same project.

7.4. What is the cost of being verifiable?

Verifier evaluations are cheap in the cases that matter: an energy-conservation check is one reduction over the force buffer; a momentum-conservation check is three. The expensive cases are exactly the ones a researcher *wants* to pay for — for example, a checker that compares end-to-end RNA distance against a reference trajectory. Even there, the cost lives in a separate process from the inner loop. Empirically, in the demos accessible from the project landing page (de Witte, 2026), verifier overhead is in the single-digit percent of frame time and disappears entirely in headless agent runs where the renderer is not attached.

8. Related work

The framework sits at the intersection of three otherwise disjoint literatures, and it is worth saying explicitly what it borrows.

Differentiable and graph-structured simulators. A line of work over the last decade has demonstrated that physics simulators can be made end-to-end differentiable, either by writing the integrator in JAX or PyTorch (Schoenholz and Cubuk, 2020; Freeman et al., 2021), by compiling domain-specific languages to differentiable kernels (Hu et al., 2019; Macklin, 2022; NVIDIA Corporation, 2023), or by parameterizing the dynamics with a learned graph network (Battaglia et al., 2016; Sanchez-Gonzalez et al., 2020; Pfaff et al., 2021). Our framework is differentiable for the same reason these are — the inner-loop term contributions are small pieces of code over typed arrays — but it differs in two ways. First, it commits to the Lagrangian, not the equations of motion, as the user-facing object; this is what makes the cross-domain story work and what makes dimensional checking trivial. Second, it adds the verifier layer; the named differentiable simulators do not, at the time of writing, ship a proof-obligation-discharge pipeline as a first-class citizen.

Unified solvers in computer graphics and animation. The graphics community has converged independently on a related "one solver, many phenomena" position, most visibly in Position-Based Dynamics (Müller et al., 2007) and its descendants, which model rigid bodies, cloth, fluids, and deformable solids as a single constraint projection over a shared particle set. Our framework makes a different commitment — Lagrangians and the equations of motion they generate, rather than positional constraints — and is therefore better suited to settings where conservation, units, and provenance matter (molecular dynamics, dark-matter constraints, biopolymer free energies) at the cost of some of PBD's ergonomic compactness for visual-effects work. The two approaches are not in opposition; PBD can be expressed as one of the solver programs of Section 3, and we encourage that reading.

Variational integrators. The mathematical content of treating the Lagrangian, not the equations of motion, as the primary discrete object has been developed extensively under the heading of discrete mechanics and variational integrators (Marsden and West, 2001; Hairer et al., 2006). The integrators that ship with the reference implementation (velocity Verlet, symplectic Euler, RATTLE, AVBD) are instances of this theory; the contribution of the present paper is not the theory but the framework that makes it routine to write down a Lagrangian, pick an integrator that respects it, and have a runtime that gates the choice against declared invariants.

Verified scientific computing. At the other extreme, there is a long tradition of using proof assistants (de Moura and Ullrich, 2021; The mathlib Community, 2020) to formalize parts of mathematics and, increasingly, parts of physics. A scene's `@expect` clauses are intentionally analogous to Lean propositions, and the verifier kernel is intentionally analogous to a Lean checker. The framework does not attempt to compete with Lean on what Lean is good at (full formal proof of a theorem); it attempts to import Lean's *architectural* discipline (small audited core, independent checker, propositions and witnesses as separate artifacts) into a numerical simulator. The two are complementary: one can imagine a scene whose verifier obligations are themselves discharged by Lean tactics, with the simulator producing the numerical witnesses that the tactic consumes.

Agentic AI for science. A third recent thread is the construction of agent harnesses for mathematical and scientific research, beginning with Polu and Sutskever's (Polu and Sutskever, 2020) GPT-f, continuing through AlphaProof and related provers, and most recently DeepMind's AI co-mathematician (Zheng et al., 2026), which embeds a hierarchical multi-agent system in a stateful workspace and shows that this style of collaboration can solve problems on FrontierMath benchmarks that single-shot LLMs cannot. Our motivation is parallel. An agent that hypothesizes a modification to a Lagrangian (a new Yukawa term, a constraint, an alternative integrator) needs an execution environment that does three things at once: it runs the modification, it produces machine-readable evidence about whether the run respected the declared physics, and it does so in a vocabulary the agent can compose with the rest of its workspace. The framework is our attempt to build exactly such an environment for physics, in the same sense in which the AI co-mathematician's working paper and ResearchState are an environment for mathematics. In practice, this means the reference runtime ships as both a CLI binary and an MCP server (Anthropic, 2024), so an agent harness (such as Claude Code) can list scenes, mutate the Lagrangian, run the kernel, and read back the verifier's PASS/FAIL through the same protocol it uses for any other tool. The companion agent *Albert* uses this loop to propose and reject modifications to scenes against observational targets.

What is not here. We are not proposing a new integrator, a new contact solver, or a new neural force field. The arithmetic in the inner loop is conventional; the contribution is the typing discipline around it that makes the arithmetic safely composable across domains and safely auditable by agents and humans.

9. Limitations and open problems

The framework is incomplete in concrete and useful ways.

Fields are first-class only in the small. Continuum fields can be represented as a fine discretization (the dam-break fluid scene is 441 particles with bond tension), but a proper finite-element or spectral-method treatment is not yet a citizen of Kind. Adding it requires an edge variant carrying a basis-function specification and a kernel-side traversal over basis-function pairs; well-defined but not done.

Constraints beyond holonomic are uneven. Bilateral and unilateral non-smooth constraints are handled via AVBD or projective dynamics; differential-algebraic systems with rank-changing constraints are not handled in a satisfactory way. A general DAE solver would be one more "solver program over the graph"; we have not yet written it.

Verifier coverage is sparse. The library covers conservation laws (energy, momentum, angular momentum, charge), dimensional consistency, and a small number of scene-specific stability checks. Fluctuation-dissipation relations, detailed balance for Monte Carlo, and PPN-style comparisons against general relativity are missing. All are just propositions, but the library is far from complete.

Soundness of generated verifiers. The Noether-style derivation of conservation laws from declared symmetries is implemented for translational and rotational invariance, but the derivation

itself is not machine-checked. A generated verifier inherits the trust of the symbolic manipulation that produced it. Pushing that manipulation through a proof assistant is an obvious next step.

Cost-model exposure. The planner's choice of traversal (cell list size, GPU launch geometry, broad-phase) is not visible to the scene author except through profiling. A framework that asks the author to think in Lagrangians should arguably also surface the cost model in Lagrangian terms; we do not yet have a clean proposal for that.

10. Conclusion

The Lagrangian formulation is the only piece of mathematical machinery that runs all the way from a freshman mechanics problem to a Standard Model gauge sector. We have argued that, taken seriously as a *computational* object rather than as analytical shorthand, it admits a triple — typed graph, single kernel, independent verifier — that suffices as the framework for a physics engine spanning the domains that today require separate codes. The construction is small enough to be reproduced and disciplined enough to be checked. We have built and shipped a reference implementation (`lagra-core` (de Witte, 2026)) that demonstrates the triple on half a dozen historically incompatible domains and we have outlined the open problems that the framework inherits from the harder corners of classical and quantum physics.

The motivating audience is not only the researcher who wants a less fragmented simulator. It is also the autonomous agent that will, in the near future, propose modifications to Lagrangians on its own. Recent work on agentic-AI workbenches for mathematics (Zheng et al., 2026) has shown that this style of system is dramatically more effective when the execution environment produces native, structured, verifiable artifacts rather than chat logs. Physics deserves the same. The Lagrangian, encoded as a graph and gated by a checker, is the artifact we propose.

Acknowledgements

This work would not exist without the years of conversations, counter-examples, and quiet patience of the wider Lagra community. The architectural debt to Lean (de Moura and Ullrich, 2021) should be obvious; the architectural debt to the AI co-mathematician paper (Zheng et al., 2026) is acknowledged with equal directness. Albert, a physics research agent built on the framework described here, has acted as a candid first reader.

References

Anthropic. Model Context Protocol specification. https://modelcontextprotocol.io/, 2024. Open protocol for connecting agentic harnesses (*e.g.* Claude Code) to external tools and data sources.

Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems 29*, 2016.

Jay P. Boris. Relativistic plasma simulation—optimization of a hybrid code. In Jay P. Boris and R. A. Shanny, editors, *Proceedings of the Fourth Conference on Numerical Simulation of Plasmas*, pages 3–67, Washington, D.C., November 1970. Naval Research Laboratory. https://apps.dtic.mil/sti/citations/ADA023511.

Anka He Chen, Ziheng Liu, Yin Yang, and Cem Yuksel. Vertex block descent. *ACM Transactions on Graphics (SIGGRAPH)*, 43(4):Article 116, 1–16, 2024. doi: 10.1145/3658179.

Anka He Chen, Ziheng Liu, Yin Yang, and Cem Yuksel. Augmented vertex block descent. *ACM Transactions on Graphics (SIGGRAPH)*, 44(4):1–12, 2025. doi: 10.1145/3731195. Augmented-Lagrangian extension of VBD for hard constraints and rigid-body contact.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

Pim de Witte. Lagra: A differentiable physics kernel for verifiable domain operations. `https://github.com/PimDeWitte/lagra`, 2026. Project landing page; live in-browser demos of the action-kernel substrate described in this paper.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax—a differentiable physics engine for large scale rigid body simulation, 2021. arXiv:2106.13281.

Herbert Goldstein, Charles Poole, and John Safko. *Classical Mechanics*. Addison-Wesley, 3rd edition, 2002.

Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. Springer Series in Computational Mathematics, vol. 31. Springer, 2nd edition, 2006.

Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato. Gradual Tensor shape checking, 2022. arXiv:2203.08402.

Adam V. Higuera and John R. Cary. Structure-preserving second-order integration of relativistic charged particle trajectories in electromagnetic fields. *Physics of Plasmas*, 24(5):052104, 2017.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 38(6), 2019.

Miles Macklin. Warp: A high-performance Python framework for GPU simulation and graphics. NVIDIA, 2022.

Jerrold E. Marsden and Matthew West. Discrete mechanics and variational integrators. *Acta Numerica*, 10:357–514, 2001.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007. Workshop on Virtual Reality Interactions and Physical Simulations (VRIPHYS 2006).

Emmy Noether. Invariante Variationsprobleme. *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, pages 235–257, 1918.

NVIDIA Corporation. NVIDIA Modulus: Open-source framework for physics-ML. `https://developer.nvidia.com/modulus`, 2023.

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W. Battaglia. Learning mesh-based simulation with graph networks. In *International Conference on Learning Representations (ICLR)*, 2021.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020. arXiv:2009.03393.

Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W. Battaglia. Learning to simulate complex physics with graph networks. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, 2020.

Samuel S. Schoenholz and Ekin D. Cubuk. JAX, M.D.: A framework for differentiable physics. *Advances in Neural Information Processing Systems*, 33, 2020.

Tim Strang, Isabella Caruso, and Sam Greydanus. Nature's cost function: Simulating physics by minimizing the action, 2023. arXiv:2303.02115.

The mathlib Community. The lean mathematical library. In Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020), 2020.

Jean-Luc Vay. Simulation of beams or plasmas crossing at relativistic velocity. *Physics of Plasmas*, 15(5):056701, 2008.

Daniel Zheng, Ingrid von Glehn, Yori Zwols, Iuliya Beloshapka, Lars Buesing, Daniel M. Roy, Martin Wattenberg, Bogdan Georgiev, Tatiana Schmidt, Andrew Cowie, Fernanda Viegas, Dimitri Kanevsky, Vineet Kahlon, Hartmut Maennel, Sophia Alj, George Holland, Alex Davies, and Pushmeet Kohli. AI co-mathematician: Accelerating mathematicians with agentic AI, 2026. arXiv:2605.06651.

A. Code references in the reference implementation

For readers who want to read the implementation behind the construction. All paths are relative to the root of the `lagra-core` crate (de Witte, 2026).

Concept (paper §)	**File path in `lagra-core`**
Lagrangian graph (§2)	`src/world/graph/lagrangian_graph.rs`
Action kernel / `ActionGraph` (§3)	`src/world/graph/kernel/ action_graph/evaluator.rs`
Solver executor (§3)	`src/world/graph/kernel/ solver_executor.rs`
Action descent (§3)	`src/world/graph/action.rs`
AVBD bridge (§3)	`src/world/physics/forces/avbd.rs`
Per-timestep root (§3)	`src/world/graph/timestep_root.rs`
Verifier set, `@expect` parser (§4)	`crates/lagra-lang/src/ir.rs`
IDE `V(q)` debug panel	`crates/lagra-ide/src/renderer/ components/inspector/ LagrangianGraph.tsx`