Bit Provenance, Graph-IR Compiling, Physics-Aware Simulation, and a Modular Neural Network Framework for Python

Executive Summary

This project is a **research harness** for building and comparing geometric/physics-aware learning systems without tying everything to a heavyweight runtime. It offers (a) a backend-agnostic tensor layer; (b) a graph IR that compiles OOP code and dataflow into scheduled DAGs; (c) a physics-based discrete-time system ("dt_system") for step-wise, adaptive simulation; (d) ASCII and OpenGL renderers for rapid truth-checking; and (e) a path toward **bit-level provenance** and **universal translation** via a minimal operator set. Most components are **theoretical prototypes**—useful, inspectable, and intended for small, reproducible demonstrations rather than production.

---

1) Rationale: Decouple Exploration from Heavy Runtimes

Typical stacks couple research to a specific framework/hardware footprint. Here, the **AbstractTensor** layer targets **pure Python**, **NumPy**, **Torch**, **JAX**, and experimental **C/OpenGL** routes—so you can explore ideas with small local demos and, when needed, switch to GPU or cloud without rewriting the system.^[1][2] Backends are swappable at module boundaries, which keeps the experiments coherent across the codebase.

---

2) Architecture at a Glance

- **Tensor & Autograd Core** — a backend-agnostic tensor API with a "whiteboard-style" VJP engine to collect residuals and compute vector-Jacobian products across ticks. This favors **inspectability** and **deterministic debugging** over black-box speed.^[3][4]
- **dt_system** — a **discrete-time**, **adaptive super-stepping** loop: multiple simulators progress within a frame using micro-steps bounded by local error/ε controls. This supports **step-wise continuity**: the learning loop sees stable, per-tick states even when internal modules contain discontinuities or mixed rates.^[5][6]
- **Graph IR & Transmogrifier** — OOP constructs are **lowered** to a graph IR, scheduled as a DAG, and, when useful, embedded in a **physical relaxation** metaphor (springs/repulsors) to find efficient orderings or allocations. This is the "universal translator" path: **bit-level primitives → IR → scheduled program**.^[7][8][9]
- **Rendering — ascii_diff** (engine) + **ascii_render** (demo) for terminal-first visualization, plus OpenGL for richer views; both connect to the same state dictionaries for trustworthy introspection.^[10][11]
- **Bit-Provenance** — a design track that traces **how bits flow and mix** through operators,

enabling “digital fluorescence” (provenance tags) and, later, energy-aware or cache-aware compilation strategies.^[12][13]

---

3) Backends and Deployment

The framework can run on:

- **Pure Python** (slow but fully transparent)
- **NumPy** and **Torch** (fast prototyping; Torch for GPU)
- **JAX** (cloud-friendly transformations)
- **OpenGL** shader paths for select rendering/simulation loops^[10]
- **C via CFFI** with a planned **Zig-assisted** build for a compact dynamic library (work-in-progress)^[2]
- A longer-term **SSA-compiled** route for fully static binaries^[8]

This mix supports **cheap, fluent exploration**: start small on CPU, switch backends when necessary, and keep the same graph-level contracts.

---

4) Bit-Provenance and the Minimal Operator Set

The compiler stack distinguishes several levels of description:

- **OOP → SSA → Scheduled DAG → Minimal Operator Set**.
  The “minimal set” is where **universal translatability** emerges: a small repertoire of **bit-level primitives** (e.g., add/sub/mul/xor, shifts, masks) that can be reliably mapped onto any backend or device.^[7][9] With that, you can **skim/simplify/unroll** at each lowering stage to optimize for speed, memory, or energy, and still maintain **provenance** (which bit/region influenced which result). This enables **program triage** (which parts must be accurate/fast) and **energy-aware policies** down to the bit flip level.^[12]

---

5) Physics-Aware System Relaxation (for Efficiency and Continuity)

Rather than baking a single global Δt, **dt_system** lets each simulator advance at its **own micro-step** within a frame, guided by **error metrics** and **step controllers** (e.g., PI-style dt control).^[5][6] This eases the pain of discontinuities:

- **Step-wise continuity for NN training**: if an internal module has hard thresholds or mixed-rate behavior, the controller can subdivide that sub-step until its state is numerically consistent. The outer learning loop still sees a coherent per-tick state trajectory—so gradients computed with the whiteboard VJP remain meaningful, even when the internals are not strictly smooth.
- **Differentiable “stand-in” signals**: physics nodes can act as **surrogate functions** (n-D

differentiable approximations) where analytic gradients are hard or missing.

A working demonstration of this posture shows up in **Transmogrifier**: take a program graph, bind it to a **spring/repulsor** layout in a spherical domain, and let relaxation reveal efficient schedules or co-locations; encode those back into the compiler for cache-friendly execution.^[8][14]

---

6) From Simple to Complex Neural Networks—Graph-Native

Neural layers here are **graph-first** modules: small linear blocks, PCA-like adapters, and activation layers that plug into AbstractTensor and the autoautograd tape. Hooks allow per-layer logging/visual panels, and rendering is pluggable (terminal or OpenGL) for fast **concept-to-screen** cycles.^[15][16]

A targeted research direction is **geometrically aware layers** (e.g., Riemannian or Laplace-Beltrami-guided operators) that **avoid lossy projections** by operating **in the manifold** rather than flattening to 2D/3D first. This area is intentionally fluid and rapidly prototyped.

---

7) File-Structure Highlights (What lives where)

- `docs/` — design notes (e.g., fused program IR, DT-graph design, abstract NN hooks). Good starting points for reviewers.^[17]
- `src/common/`
    - `double_buffer/` — ping-pong and worker-safe buffering primitives (demos included).^[18]
    - `dt_system/` — **dt_graph**, **engine_api**, **dt_controller**, **dt_solver**, **spectral_dampener**, and mechanics/integrator subpackages. This is the **time-stepping backbone**.^[5][6][19]
    - `tensors/` — AbstractTensor, autoautograd (whiteboard runtime), backends, and NN building blocks; design docs (`EXPLAINER.md`, etc.).^[1][3][15]
    - `quad_buffer/` and `index_composer/` — supplemental buffering/indexing utilities.^[20][21]
- `src/rendering/`
    - `ascii_diff/` — the **actual ASCII engine**: framebuffer, kernel classifier, presets, demos (clock/benchmark).^[10]
    - `ascii_render/` — thin **demo wrapper** around ascii_diff.^[11]
    - `opengl_render/` — API and threaded renderer stubs for GL-based visualization.^[22]
- `src/compiler/`
    - `bitops.py` & `bitops_translator.py` — **bit-level primitives** and their translation to graph nodes; supports the “minimal operator set” route.^[7][9]

    - `ssa_builder.py`, `process_graph_helper.py` — lowering from higher-level forms to **SSA** and **scheduled graphs**.^[8]
- `src/transmogrifier/`
    - `graph/` — graph builders and compilers: **BitTensorMemoryGraph**, **ProcessGraph**, **GraphDeepCompiler**.^[8][14]
    - `cycle_unroller.py`, `operator_defs.py`, `solver_types.py` — loop unrolling, operator signatures, and core node/edge types for schedulers.^[22][23][24]
- `src/bitbitbuffer/` — bit-level ring buffers, indexing, masking, and helpers; foundation for provenance.^[25]
- `tests/` — extensive but evolving coverage across autograd, dt_system, rendering, and compiler subsystems. Expect some failures during refactors.

Reality check: Transmogrifier "almost works"—its compilers and schedulers largely run, but **correctness** depends on **cell-sim/dt_system** semantics stabilizing. Likewise, ascii_render defers to ascii_diff for real work.

---

8) Current Status and Risks

- **Maturity**: prototypes across the board; many modules are intentionally partial.
- **Interdependence**: changes must be **meticulously propagated**.
- **Backends**: C/GL paths are **promising** but **incomplete**; Torch/JAX remain the most practical compute engines today.
- **Autograd**: the whiteboard VJP is ideal for **small, inspectable** runs; large-scale performance depends on future backends and optimized tapes.

Footnotes (Repository Pointers)

[1] `src/common/tensors/README.md` — backend-agnostic goals and interfaces.
[2] `src/common/tensors/accelerator_backends/c_backend.py` + `.../c_backend/AGENTS.md` — CFFI design and Zig-assisted build stubs.
[3] `src/common/tensors/autoautograd/whiteboard_runtime.py` — whiteboard VJP, residual/tape mechanics.
[4] `src/common/tensors/backward_registry.py` — backward op registrations and policies.
[5] `src/common/dt_system/dt.py` — dt loop primitives and integration points.
[6] `src/common/dt_system/dt_controller.py` — adaptive/PI step controllers and dt scaling.
[7] `src/compiler/bitops.py` — minimal bit-level operator repertoire.
[8] `src/compiler/ssa_builder.py`, `src/compiler/process_graph_helper.py` — IR lowering and SSA construction.

[9] `src/compiler/bitops_translator.py` — mapping from bit ops to graph primitives.
[10] `src/rendering/ascii_diff/` — framebuffer, kernel classifier, presets, demos (e.g., `clock_demo.py`, `benchmark_demo.py`).
[11] `src/rendering/ascii_render/` — demo wrapper (`demo_ascii_render.py`).
[12] `src/bitbitbuffer/` + `src/bitbitbuffer/helpers/` — buffers, indexers, slices, streams (provenance foundations).
[13] `docs/FUSED_PROGRAM_IR.md` — notes toward a common IR surface.
[14] `src/transmogrifier/graph/graph_deep_compiler.py`, `graph_express2.py`, `graph/memory_graph/` — graph builders and deep compiler.
[15] `src/common/tensors/abstract_nn/` — small NN blocks (`linear_block.py`, `pca_layer.py`) and adapters.
[16] `src/common/tensors/abstract_nn/hooks.py` — hook panel and training-time events.
[17] `docs/dt_graph_design.md`, `docs/abstract_nn_hooks.md` — high-level design notes.
[18] `src/common/double_buffer/` — base/core/lock/workers and demos.
[19] `src/common/dt_system/engine_api.py`, `dt_graph.py`, `dt_solver.py`, `spectral_dampener.py` — orchestration/graphing.
[20] `src/common/quad_buffer/` — `quad_buffer.py`, `tribuffer.py`.
[21] `src/common/index_composer/indexcomposer.py`.
[22] `src/rendering/opengl_render/` — API, threaded renderer stubs.
[23] `src/transmogrifier/cycle_unroller.py` — loop unrolling strategies.
[24] `src/transmogrifier/operator_defs.py`, `src/transmogrifier/solver_types.py` — operator signatures and node/edge types.
[25] `src/bitbitbuffer/helpers/bitbitindex.py`, `.../bitbitslice.py`, `.../rawspan.py`, etc.

Appendix: Notes on "Step-Wise Continuity"

Why it helps: Many real systems are **piecewise** or **event-driven**. For training, it's enough that *each outer tick's state* lies on a consistent, low-error manifold—**even if** the tick internally required 3→30 micro-steps. The **dt controller** contracts/inflates those micro-steps until local criteria are satisfied; the **autograd tape** then differentiates through the **sequence of stable sub-states**, insulating parameter updates from internal discontinuities. This is the practical bridge between **clean gradients** and **messy simulators**.

Sources & Selected Readings (Annotated)

Purpose. This page curates key, peer-reviewed (and a few influential preprint) works that ground the project's aims: geometry-aware learning, graph-native representations, physics-structured training, operator learning, and constraint-aware layers. Each entry has a brief, plain-language note on what it contributes and how it informs this framework.

A. Field Overviews (context for geometry in ML)

[1] Bronstein, Bruna, Cohen, Veličković. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges* (2021).
What it says: A broad map of how modern ML should respect the structure of data—symmetries, graphs, and manifolds—not just raw arrays.
Why it matters here: Validates a design where geometry is first-class and modules stay portable across backends while preserving structure.

[2] Bronstein, Bruna, LeCun, Szlam, Vandergheynst. *Geometric Deep Learning: Going Beyond Euclidean Data* (2016/2017).
What it says: Early, influential survey of learning on graphs and manifolds.
Why it matters here: Motivates a graph-native IR and manifold-aware layers as default building blocks, not afterthoughts.

B. Symmetry-Respecting Models (equivariance)

[3] Weiler & Cesa. *General E(2)-Equivariant Steerable CNNs* (NeurIPS 2019).
What it says: Convolutional layers that "do the right thing" under 2D rotations/reflections by design.
Why it matters here: Template for symmetry-aware kernels that plug into a unified tensor API.

[4] Finzi, Stanton, Izmailov, Wilson. *LieConv* (ICML 2020).
What it says: Convolutions equivariant to continuous Lie groups; one recipe across many symmetry groups.
Why it matters here: Encourages a small set of primitive operations the IR can preserve while retargeting to different domains.

[5] Cohen, Weiler, Kicanaoglu, Welling. *Gauge-Equivariant CNNs & Icosahedral CNN* (ICML 2019).
What it says: Layers that are consistent on curved domains (spheres/meshes) via local frames ("gauges").
Why it matters here: Direct precedent for manifold-native layers that run on spherical/mesh data without lossy flattening.

[6] Fuchs, Worrall, Fischer, Welling. *SE(3)-Transformer* (NeurIPS 2020).
What it says: Self-attention that's rigid-motion equivariant in 3D.
Why it matters here: A path toward attention modules that stay physically consistent in 3D scenes.

[7] Thomas et al. *Tensor Field Networks* (2018, preprint).
What it says: Uses spherical harmonics to handle vectors/tensors with rotation awareness.
Why it matters here: Connects natural "typed" features (scalars/vectors) to spectral methods used

elsewhere in the stack.

[8] Satorras, Hoogeboom, Welling. *E(n)-Equivariant GNNs* (ICML 2021).
What it says: Lightweight equivariant message-passing without heavy tensor representations.
Why it matters here: A pragmatic default for symmetry-aware graph layers.

[9] Cohen, Geiger, Weiler. *A General Theory of Equivariant CNNs on Homogeneous Spaces* (NeurIPS 2019).
What it says: A unifying mathematical frame for equivariant CNNs.
Why it matters here: Helps specify what the compiler must not break when lowering high-level models to IR.

[10] Batzner et al. *NequIP* (*Nature Communications* 2022).
What it says: Strong accuracy and data efficiency using E(3)-equivariant networks in atomistic modeling.
Why it matters here: Evidence that symmetry priors reduce data needs—useful when experiments are small or expensive.

C. Physics-Structured Learning (conservation & mechanics)

[11] Greydanus, Dzamba, Yosinski. *Hamiltonian Neural Networks* (NeurIPS 2019).
What it says: Learn a Hamiltonian so rollouts conserve energy/momentum by construction.
Why it matters here: Shows how architectural structure can stabilize learning in dynamical systems.

[12] Cranmer et al. *Lagrangian Neural Networks* (ICLR 2020 workshop/related).
What it says: Learn a Lagrangian and derive dynamics from it.
Why it matters here: Complements the above with an alternative, principled route to physics-aware layers.

[13] Lutter, Ritter, Peters. *Deep Lagrangian Networks (DeLaN)* (ICLR 2019).
What it says: Enforce Euler-Lagrange structure for better generalization in control tasks.
Why it matters here: Signals how to combine structure with gradient-based training in practice.

D. Operator Learning & PDE Surrogates

[14] Raissi, Perdikaris, Karniadakis. *Physics-Informed Neural Networks (PINNs)* (*J. Comput. Phys.* 2019).
What it says: Use PDE residuals directly in the loss to guide learning.
Why it matters here: Lets data and physics work together when ground truth is scarce.

[15] Lu et al. *DeepONet* (*Nature Machine Intelligence* 2021).
What it says: Learn mappings between function spaces (operators), not just numbers.
Why it matters here: A flexible way to drop in learned "mini-solvers" beside traditional simulators.

[16] Kovachki et al. *Neural Operator* (*JMLR* 2023).
What it says: A comprehensive treatment of operator learning with discretization-invariant goals.
Why it matters here: Theory scaffolding for portable operator blocks across meshes/resolutions.

[17] Li et al. *Fourier Neural Operator (FNO)* (NeurIPS 2021; arXiv 2020).
What it says: Learn PDE solution maps in Fourier space; fast and mesh-agnostic.

Why it matters here: Aligns with spectral/FFT features and offers a strong candidate kernel for future acceleration.

E. Learned Simulators on Graphs/Meshes

[18] Sanchez-Gonzalez et al. *Learning to Simulate with Graph Networks* (ICML 2020).
What it says: Particle/graph-based simulators trained directly from rollouts.
Why it matters here: A natural fit for graph IR and multi-rate stepping.

[19] Pfaff et al. *MeshGraphNets* (2020, arXiv/ML4Eng).
What it says: Mesh-aware GNNs for simulation with adaptive discretization.
Why it matters here: Blueprint for mesh-domain lowering and visual verification.

F. Optimization as a Layer (hard constraints inside nets)

[20] Amos, Xu, Kolter. *Input-Convex Neural Networks* (ICML 2017).
What it says: Architectures guaranteed convex in the right variables.
Why it matters here: Stable "sub-steps" you can embed where guarantees help.

[21] Amos, Kolter. *OptNet* (ICML 2017).
What it says: Differentiate through a quadratic program—optimization as a layer.
Why it matters here: Insert exact constraints without leaving end-to-end training.

[22] Agrawal et al. *CVXPY Layers* (NeurIPS 2019).
What it says: Disciplined convex programs turned into differentiable layers.
Why it matters here: A practical toolchain for constraint-aware components.

Full Citations

1. Bronstein MM, Bruna J, Cohen T, Veličković P. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. 2021.
2. Bronstein MM, Bruna J, LeCun Y, Szlam A, Vandergheynst P. *Geometric Deep Learning: Going Beyond Euclidean Data*. 2016/2017.
3. Weiler M, Cesa G. *General E(2)-Equivariant Steerable CNNs*. NeurIPS, 2019.
4. Finzi M, Stanton S, Izmailov P, Wilson AG. *Generalizing CNNs for Equivariance to Lie Groups*. ICML, 2020.
5. Cohen TS, Weiler M, Kicanaoglu B, Welling M. *Gauge-Equivariant CNNs and the Icosahedral CNN*. ICML, 2019.
6. Fuchs FB, Worrall DE, Fischer V, Welling M. *SE(3)-Transformers*. NeurIPS, 2020.
7. Thomas N, Smidt T, Kearnes S, et al. *Tensor Field Networks*. 2018 (preprint).
8. Satorras VG, Hoogeboom E, Welling M. *E(n) Equivariant Graph Neural Networks*. ICML, 2021.

9. Cohen TS, Geiger M, Weiler M. *A General Theory of Equivariant CNNs on Homogeneous Spaces*. NeurIPS, 2019.
10. Batzner SL, Musaelian A, Sun L, et al. *E(3)-equivariant GNNs for Interatomic Potentials*. *Nature Communications*, 2022.
11. Greydanus S, Dzamba M, Yosinski J. *Hamiltonian Neural Networks*. NeurIPS, 2019.
12. Cranmer M, Greydanus S, Hoyer S, et al. *Lagrangian Neural Networks*. ICLR 2020 workshop/related.
13. Lutter M, Ritter C, Peters J. *Deep Lagrangian Networks*. ICLR, 2019.
14. Raissi M, Perdikaris P, Karniadakis GE. *Physics-Informed Neural Networks*. *Journal of Computational Physics*, 2019.
15. Lu L, Jin P, Pang G, Zhang Z, Karniadakis GE. *DeepONet*. *Nature Machine Intelligence*, 2021.
16. Kovachki NB, Li Z, Liu B, et al. *Neural Operator. JMLR*, 2023.
17. Li Z, Kovachki NB, Azizzadenesheli K, et al. *Fourier Neural Operator*. NeurIPS 2021 (arXiv 2020).
18. Sanchez-Gonzalez A, Godwin J, Pfaff T, et al. *Learning to Simulate with Graph Networks*. ICML, 2020.
19. Pfaff T, Fortunato M, Sanchez-Gonzalez A, Battaglia P. *Learning Mesh-Based Simulation with Graph Networks*. 2020 (arXiv/ML4Eng).
20. Amos B, Xu L, Kolter JZ. *Input-Convex Neural Networks*. ICML, 2017.
21. Amos B, Kolter JZ. *OptNet: Differentiable Optimization as a Layer in Neural Networks*. ICML, 2017.
22. Agrawal A, Amos B, Barratt S, et al. *Differentiable Convex Optimization Layers*. NeurIPS, 2019.

How to read this page: Start with A/B to see why the system is geometry-native; jump to C/D for physics and operators if you're coming from simulation; visit F when constraints/safety are central.