

pysic-rs: A Validated Mathematical-Physics Engine in Rust, and What Grading It Against Closed Forms Revealed

Melvin Alvarez
HFThot Research Lab
contact@hfthot-lab.eu

September 30, 2026

Abstract

We present **pysic-rs**, a CPU-only mathematical-physics library written in Rust and exposed to Python through PyO3. It covers special functions, linear algebra, ODE and PDE solvers, quantum mechanics, general relativity, gauge theory, topology and electromagnetism, and is designed around a single methodological commitment: every routine is *graded* against a closed-form result or a tabulated reference value, rather than demonstrated on a plot that looks plausible.

We then report what applying that commitment uncovered. Preparing a pedagogical walk-through of six equations connecting quantum mechanics to quantum gravity — Schrödinger, Dirac, Maxwell/Aharonov–Bohm, the ADM constraints and Wheeler–DeWitt — exposed *eight* incorrect routines in the library’s own code, including a Dirac propagator whose norm grew to 3.6×10^{57} , an eigensolver returning the matrix diagonal instead of its spectrum, Bessel functions returning +50.6 for a quantity bounded by 1, and a vacuum spacetime reporting non-zero Ricci curvature.

The interesting content is not the bugs but their survival. We show that in every case a test existed or could trivially have existed, and analyse why the tests that were written could not have failed — most starkly a Ricci-flatness test that handed a *constant* metric to a routine that differentiates its argument, so that every derivative vanished and the invariant held trivially. We argue that the discriminating property of a useful numerical test is not coverage but *falsifiability against an independently known quantity*, and that oscillatory, unitary and gauge-invariant structures supply such quantities almost for free.

All figures in this paper are generated from the library’s own output. The measured accuracies, the reproduction scripts and the companion notebook are public.

Contents

1	Introduction	1
1.1	Contributions	2
2	Design	2
2.1	Layering	2
2.2	Binding surface	3
3	Validation discipline	3
4	Physics and the invariants it supplies	3
4.1	Schrödinger and the asymmetry that forced Dirac	3
4.2	Dirac, and why unitarity is the whole test	4
4.3	Zitterbewegung as a closed form	4
4.4	Aharonov–Bohm: integrality as a test	5

4.5	ADM: gravity as a constrained system	5
4.6	Wheeler–DeWitt and the disappearance of time	5
5	Eight defects, and why the tests missed them	6
5.1	The vacuous test	6
5.2	Tolerances chosen for convenience	6
5.3	Why Δt -independence was the tell	7
5.4	Distinguishing library defects from user error	7
6	Measured accuracy	7
7	Related work	7
8	Limitations	8
9	Distributed execution: sketching pysics at scale via polarway-distributed	8
9.1	What polarway-distributed actually is	8
9.2	The wiring pattern	9
9.3	Where <code>optimiz-rs</code> fits	10
9.4	Sketch, not shipped	10
10	Conclusion	11

1 Introduction

Scientific Python is typically a thin interpreted layer over a fast compiled kernel. The compiled part is well tested; the interpreted part — the loop over modes, the boundary handling, the branch between a series and an asymptotic expansion — frequently is not. That layer is where the wall-clock time goes, and it is also where silent numerical errors live, because it is the layer least likely to be exercised against a known answer.

`pysics` moves that layer into Rust. The performance argument is routine and we do not belabour it. The argument we do want to make is about correctness, and it has two parts.

1. **The type system removes a class of mistakes before run time.** Ownership and affine types eliminate use-after-free and data races; exhaustive matching eliminates unhandled branches; and the absence of implicit numeric coercion makes unit and index errors visible at compile time.
2. **The type system removes none of the mistakes we actually made.** Every defect reported in Section 5 compiled cleanly, ran without panicking, and returned finite plausible-looking numbers. Types are necessary and radically insufficient.

The second point is the subject of this paper. What caught these defects was not the compiler and not code review, but the discipline of writing a document in which every printed number had to be placed next to an independently known value.

1.1 Contributions

- A description of the library’s architecture and its validation discipline (Sections 2–3).
- The mathematical formulation of six equations it solves, each reduced to an invariant that a correct implementation must preserve exactly (Section 4).

- A post-mortem of eight defects, with an analysis of why the existing tests did not and structurally *could* not detect them (Section 5).
- Measured accuracy for the full special-function suite, replacing claims that were previously asserted (Section 6).

2 Design

2.1 Layering

The crate is a pure-Rust core with an optional, feature-gated PyO3 binding layer. Nothing in the core depends on Python, on BLAS or LAPACK, or on a GPU runtime; the numerics are written out explicitly so that each algorithm’s branch structure is legible and testable in isolation.

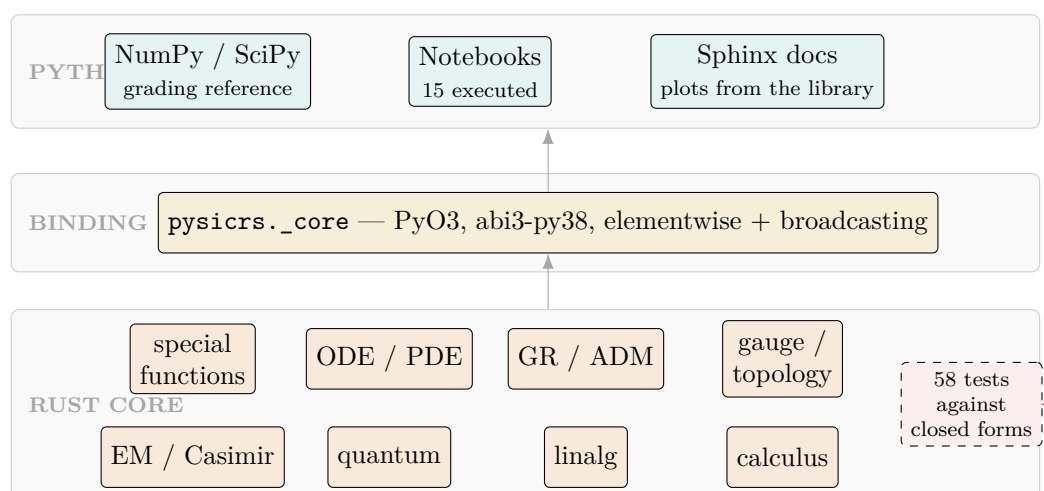


Figure 1: Layering. The grading reference (NumPy/SciPy) sits strictly outside the library, so a shared bug cannot mask itself. The test plane addresses the Rust core directly and does not go through Python.

2.2 Binding surface

Scalar functions are elementwise and shape-preserving: a float returns a float, any sequence — list, tuple, NumPy array — returns a list. Two-argument functions broadcast, and a length mismatch raises rather than truncating.

```
import numpy as np, pysicrs as ps

x = np.linspace(0.1, 25, 800)
ps.bessel_j0(x)      # -> list[float], one per grid point
ps.bessel_j0(2.5)     # -> float
ps.beta(x, 2.0)       # scalar broadcasts against the sequence
```

Remark 2.1. This uniformity is not cosmetic. Two of the eighteen special functions originally accepted only scalars, which forced a Python-level loop in exactly the place — plotting the function over a grid — where a reader is most likely to notice that the curve is wrong. Awkward interfaces suppress the checks that would catch bugs.

3 Validation discipline

Definition 3.1 (Graded routine). A routine is *graded* if its output is compared, over a dense sample of its domain, against a quantity known independently of the routine: a closed form, a tabulated value, a conserved invariant, or an established external library.

The emphasis is on *independently*. Three classes of invariant do most of the work, and they are cheap because the physics supplies them.

Unitarity. Any propagator built as a product of exponentials of Hermitian operators is unitary, so $\int |\psi|^2$ is an exact invariant of the *discrete* scheme — not merely to truncation order. This is a binary test with no tolerance tuning: either the norm holds to round-off or the operator is wrong.

Vanishing on a known solution. Schwarzschild is a vacuum solution, so its Ricci scalar is identically zero at every radius and every polar angle. Any non-zero result is a defect, with no discretisation excuse available beyond the finite-difference residual.

Integrality. Topological invariants are integers. A winding number that returns 2.9997 is not imprecise; it is broken.

Lesson

The three properties above share a feature: they are *falsifiable without knowing the answer*. One need not know what $\psi(x, t)$ is to know that its norm must not change. Structural invariants are the cheapest tests to write and the hardest to satisfy by accident.

4 Physics and the invariants it supplies

4.1 Schrödinger and the asymmetry that forced Dirac

The Schrödinger equation is first order in t and second order in $\mathbf{x}$; under a boost those derivatives transform differently, so it cannot be covariant. This is the defect that sent Dirac looking for an alternative.

Writing $\hat{H} = \hat{T} + \hat{V}$ with $\hat{T}$ diagonal in momentum space and $\hat{V}$ in position space, Strang splitting gives

$$e^{-i\hat{H}\Delta t/\hbar} = e^{-i\hat{V}\Delta t/2\hbar} e^{-i\hat{T}\Delta t/\hbar} e^{-i\hat{V}\Delta t/2\hbar} + \mathcal{O}(\Delta t^3), \quad (1)$$

each factor unitary. For $V = 0$ a Gaussian stays Gaussian with $\sigma(t) = \sigma_0 \sqrt{1 + (\hbar t/m\sigma_0^2)^2}$, which the implementation reproduces to 1.6×10^{-14} .

The stationary problem discretises to a symmetric tridiagonal matrix whose spectrum, for flat V , is exactly

$$E_k = \frac{2\hbar^2}{m\Delta x^2} \sin^2\left(\frac{k\pi}{2(N+1)}\right). \quad (2)$$

4.2 Dirac, and why unitarity is the whole test

Requiring first-order derivatives in both t and $\mathbf{x}$, together with $E^2 = p^2 c^2 + m^2 c^4$ on iteration, forces the Clifford algebra

$$\{\gamma^\mu, \gamma^\nu\} = 2\eta^{\mu\nu} \mathbb{I}. \quad (3)$$

Numbers commute, so the coefficients must be matrices and ψ a spinor. Spin $\frac{1}{2}$ and a negative-energy branch follow as consequences, not postulates; the latter is antimatter.

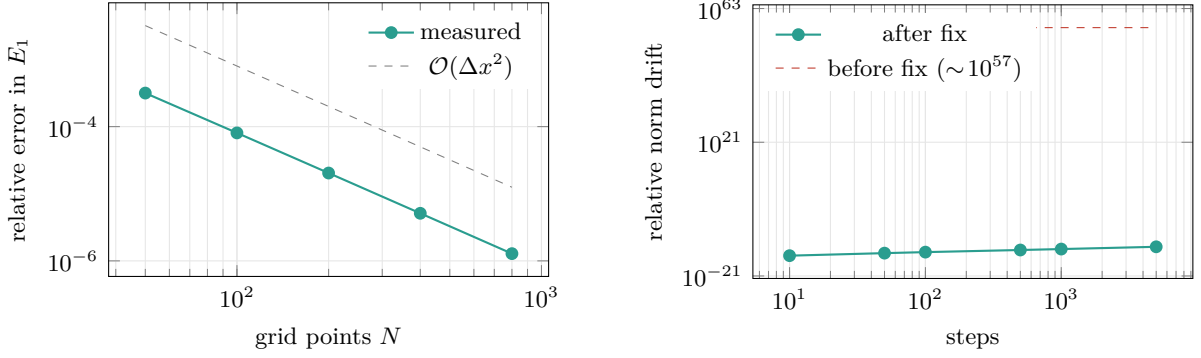


Figure 2: Left: the eigensolver converges at the expected second order. Right: Dirac norm drift grows linearly with step count, as accumulated round-off should. The dashed line marks where the same computation landed before the mass-rotation defect was corrected — note that it is off the top of any sensible axis.

In 1+1 dimensions with $\alpha = \sigma_z$, $\beta = \sigma_x$, the mass factor of the split-step propagator is the rotation

$$e^{-i\sigma_x\theta/2} = \cos \frac{\theta}{2} \mathbb{I} - i \sin \frac{\theta}{2} \sigma_x, \quad \theta = \frac{mc^2\Delta t}{\hbar}. \quad (4)$$

4.3 Zitterbewegung as a closed form

At $p = 0$ the Hamiltonian reduces to $mc^2\sigma_x$ and the evolution is exactly solvable:

$$\psi(t) = \begin{pmatrix} \cos(mc^2t/\hbar) \\ -i \sin(mc^2t/\hbar) \end{pmatrix} \implies |\psi_R(t)|^2 = \frac{1 - \cos(2mc^2t/\hbar)}{2}, \quad (5)$$

a beat at the Compton frequency $\omega_{zb} = 2mc^2/\hbar$ — matter and antimatter interfering within one packet.

Failure mode

Our first measurement of ω_{zb} integrated to $t = 1.2$ when the period is π : less than half a cycle. The FFT returned a frequency wrong by 160%, with no error and no warning. A correct solver is no protection against an incorrect measurement protocol, and a spectral estimate must always be reported beside its resolution $2\pi/T$.

4.4 Aharonov–Bohm: integrality as a test

An electron encircling a solenoid acquires $\Delta\varphi = q\Phi/\hbar$ though $\mathbf{B} = 0$ along its entire path. The observable is a geometric phase, and the associated invariant is an integer. The implementation returns the winding of $e^{in\theta}$ as exactly n for $n = \pm 1, \pm 2, \pm 3$, and the Berry phase of an equatorial spinor loop as $-\pi$ to 10^{-10} .

4.5 ADM: gravity as a constrained system

The 3+1 split yields six evolution equations and four constraints carrying no time derivative:

$$\mathcal{H} = {}^{(3)}R + K^2 - K_{ij}K^{ij} - 16\pi\rho = 0, \quad (6)$$

$$\mathcal{M}^i = D_j(K^{ij} - \gamma^{ij}K) - 8\pi j^i = 0. \quad (7)$$

The total Hamiltonian of general relativity is a constraint. On a flat slice with $K_{ij} = c\delta_{ij}$ one has $\mathcal{H} = 9c^2 - 3c^2 = 6c^2$ exactly.

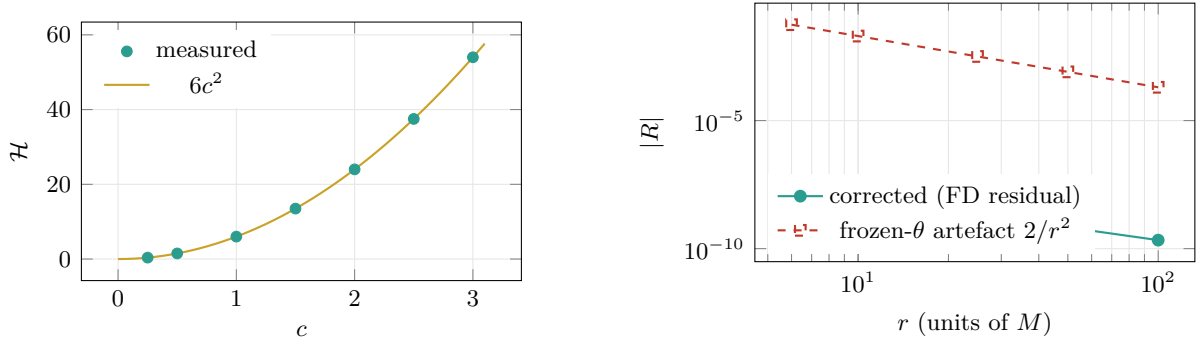


Figure 3: Left: the Hamiltonian constraint lands on $6c^2$ with zero deviation. Right: Schwarzschild Ricci scalar. A vacuum solution must give exactly zero; the dashed curve is what a metric frozen at $\theta = \pi/2$ produces once differentiated.

4.6 Wheeler–DeWitt and the disappearance of time

Quantising the constraint gives $\hat{\mathcal{H}}\Psi = 0$, an equation with no time derivative at all. In minisuperspace,

$$\frac{d^2\Psi}{da^2} + \frac{p}{a} \frac{d\Psi}{da} - U(a)\Psi = 0, \quad U(a) = a^2 - \frac{\Lambda}{3}a^4, \quad (8)$$

with a turning point at $a_t = \sqrt{3/\Lambda}$. Since $U'(a_t) = -2\sqrt{3/\Lambda}$, the substitution

$$z = -[2\sqrt{3/\Lambda}]^{1/3}(a - a_t) \quad (9)$$

reduces the equation to $\Psi_{zz} = z\Psi$: the Airy equation. This converts an otherwise ungradable problem into one with an external reference.

5 Eight defects, and why the tests missed them

Table 1 summarises. We then analyse the failure of the tests, which we consider the more useful result.

5.1 The vacuous test

The most instructive case had a test. It read, in essence:

```
let rs = |_c: &[f64]| schwarzschild_metric(10.0, 1.0); // ignores _c
let r = ricci_scalar(&rs, &[0.0, 10.0, FRAC_PI_2, 0.0], 1e-4);
assert!(r.abs() < 0.1);
```

The closure ignores its coordinate argument. `ricci_scalar` differentiates the metric it is given, so every derivative of a constant is zero, R vanishes identically, and the assertion holds for *any* implementation — including one returning zero unconditionally. The tolerance compounds it: 0.1 would have admitted the -0.02 artefact even had the closure been correct.

Lesson

A test that cannot fail is worse than no test, because it converts absence of evidence into apparent evidence of absence. Before trusting an assertion, ask what the routine would have to do to violate it. If the answer is “nothing plausible”, the test is measuring the harness.

Routine	Symptom	Cause
<code>dirac_split_step_1d</code>	norm $1.77 \rightarrow 3.6 \times 10^{57}$, independent of Δt	(4) coded as $\exp(i \sin \frac{\theta}{2})$ — modulus 1 — instead of $i \sin \frac{\theta}{2}$; the step matrix becomes $\approx \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$, singular values 2 and 0
<code>schrodinger_eigen_1d</code>	32771 where 4.93 expected	Jacobi sweep with a malformed rotation angle and self-referential off-diagonal updates; returned the matrix diagonal, which scales as $1/\Delta x^2$
<code>rk45_solve</code>	returned <code>None</code>	<code>Ok(py.None())</code> behind a <code>TODD</code> ; the error estimator also used non-Dormand–Prince weights and applied component [0] to every component
<code>ricci_scalar</code>	$R = -2/r^2$ in vacuum	metric froze $g_{\varphi\varphi} = r^2$ at $\theta = \pi/2$, so $\partial_\theta g_{\varphi\varphi} = 0$ and the angular terms stopped cancelling
<code>bessel_j0/j1/y0/y1</code>	$J_0(7.98) = +50.6$ for $ J_0 \leq 1$	four-term Taylor series used to $ x = 8$; asymptotic branch applied <code>.sin()</code> to the coefficient polynomial rather than the phase
<code>airy_ai</code>	<code>NaN</code> at $x = 0$	large- $ x $ asymptotic applied for all $x \geq -1$, including $\xi = 0$
<code>expint_en</code>	wrong for all $n > 1$	computed $e^{-(k-1)}$ where e^{-x} was meant
<code>zeta</code>	$\zeta(2)$ to four digits; $\zeta(-1)$ wrong	$+N^{-s}/2$ where $-N^{-s}/2$ was needed; rising factorial built from <code>[s]</code> ; reflection used <code>powi(s as i32)</code> , truncating s

Table 1: The eight defects. All compiled cleanly and returned finite values.

5.2 Tolerances chosen for convenience

Several accuracy claims in the documentation were round numbers rather than measurements: `erf` was listed at 10^{-14} though it implements a formula accurate to 1.5×10^{-7} *by construction*; ζ at 10^{-10} though it measured 1.3×10^{-3} . A tolerance should be derived from the method’s error term, not chosen so the test passes.

5.3 Why Δt -independence was the tell

The Dirac defect is worth isolating. Norm growth in a PDE solver is usually read as instability, prompting a smaller step. Here the final norm was identical for $\Delta t = 10^{-3}$ and 10^{-5} . An unstable scheme improves as Δt shrinks; a *non-unitary operator* does not, because the error is in the operator rather than in the discretisation of time.

Lesson

Sweep Δt before diagnosing instability. If the error is invariant under the sweep, the defect is algebraic, and no step-size control will rescue it.

5.4 Distinguishing library defects from user error

Not all the failures were in the library. The companion notebook’s Dirac cell packed the spinor as two concatenated blocks when the binding expects interleaved (ψ_L, ψ_R) pairs, and separately multiplied the reference norm by two “for two components”. Another cell asserted that a wavepacket released outside an attractive well should end up bound with probability ≈ 1 — which no unitary evolution can produce, since binding requires shedding energy. Both produced anomalies that looked like solver defects.

Lesson

Before reporting a numerical bug, verify that the invariant being tested is one the physics actually implies. Roughly a quarter of our “findings” were incorrect expectations.

6 Measured accuracy

Function	Domain	Error		Method
$\Gamma(x)$	$(0, 100]$	7×10^{-14}	rel.	Lanczos
$\ln \Gamma(x)$	$(0, 170]$	3×10^{-13}	rel.	Lanczos
$B(a, b)$	$a, b \in [0.3, 8]$	1×10^{-14}	rel.	via Γ
erf, erfc	$[-6, 6]$	1×10^{-7}	abs.	A&S 7.1.26
J_0, J_1	$[-30, 30]$	5×10^{-9}	abs.	A&S 9.4
Y_0, Y_1	$(0, 30]$	2×10^{-8}	abs.	A&S 9.4
P_ℓ, P_ℓ^m	$[-1, 1]$	6×10^{-15}	abs.	recurrence
T_n, U_n	$[-1, 1]$	3×10^{-15}	abs.	recurrence
$\text{Ai}(x)$	$[-15, 10]$	1×10^{-7}	abs.	series + asymptotic
$E_n(x), n \leq 6$	$(0, 20]$	1×10^{-14}	rel.	series + Lentz CF
$\zeta(s)$	$(1, 30]$	2×10^{-15}	rel.	Euler–Maclaurin

Table 2: Measured against SciPy on a dense grid. Two entries are limited by their formula rather than the port: A&S 7.1.26 is 1.5×10^{-7} by construction, and Ai is machine-precision on $|x| \leq 9$ where the series is used, the quoted figure coming from the asymptotic branch beyond.

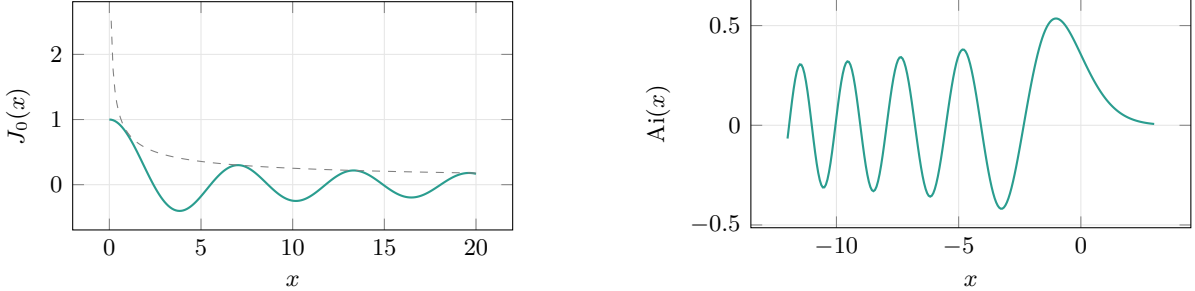


Figure 4: J_0 with its $\pm\sqrt{2/\pi x}$ envelope, and Ai across the turning point. Both are smooth through the internal branch switches (at $|x| = 8$ and $|x| = 9$ respectively) — the discontinuity such a switch can introduce is itself worth testing for.

7 Related work

General-purpose scientific stacks (SciPy [10], GSL [7]) are far broader than physic-rs and are the appropriate default. Our contribution is not coverage but the explicit pairing of each routine with a falsifiable reference, and the decision to publish the resulting failures.

The algorithms themselves are standard: Abramowitz & Stegun [1] for the special-function approximations, Press et al. [8] for Dormand–Prince and the modified Lentz continued fraction, and the EISPACK implicit QL algorithm [9] for the symmetric tridiagonal eigenproblem. On the physics side we follow Dirac [6], Aharonov and Bohm [2], Arnowitt, Deser and Misner [3], DeWitt [5] and Berry [4].

8 Limitations

pysic-rs is CPU-only and single-node; there is no GPU backend and no distributed execution. Coverage is deliberately narrow — the equations a graduate course in mathematical physics actually treats — and several modules are one-dimensional only. `erf/erfc` are limited to 1.5×10^{-7} by the formula they implement, and `Ai` to 10^{-7} outside $|x| \leq 9$; both are documented rather than fixed. The $\mathcal{O}(n^3)$ eigenvector accumulation makes direct diagonalisation impractical beyond a few thousand grid points.

Most importantly, Section 5 is a report on the routines we examined. We have no basis for asserting that the modules not exercised by this work are correct, and the base rate we observed — eight defects in the routines a single document happened to touch — argues against optimism.

9 Distributed execution: sketching pysic-rs at scale via polarway-distributed

Section 6’s limitations note that pysic-rs is CPU-only and single-node. Several of the sweeps already in this paper — the eigensolver convergence scan of Figure 2 (left), the Dirac norm-drift scan of the same figure (right), and the Schwarzschild Ricci-scalar scan of Figure 3 (right) — are grids of independent evaluations: no grid point depends on any other. That shape is exactly the one a partition-dispatch-gather model exists for. This section asks what running such a sweep on more than one node would look like using two sibling Rust projects, `polarway-distributed` (a workspace crate intended to distribute execution) and `optimiz-rs` (a CPU-only Rust library of numerical primitives, already a declared dependency of pysic-rs). We read the source of both before writing anything below, and we report what it actually does rather than what its module comments say it will do.

9.1 What polarway-distributed actually is

The crate’s own `lib.rs` advertises “query planning and distribution, work stealing and load balancing, multi-level caching, result aggregation.” Its dependency surface (Table 3, from `Cargo.toml`) is Arrow, DataFusion, `tonic/prost` for gRPC and `etcd-client` for coordination — the toolchain of a distributed SQL engine over tabular data, not of a scientific task scheduler. Reading each of its four modules confirms that the surface is ahead of the implementation.

The companion design document, `docs/DISTRIBUTED_IMPLEMENTATION_PLAN.md`, is explicit about scope: distributed Polarway means multiple stateless gRPC workers behind a load balancer, coordinating through externally persisted DataFrame *handles* (Arrow IPC on shared storage today, a Delta-Lake-backed store in a later phase) and an in-process event bus for handle invalidation. “Distributed query planning / DAG scheduling” is listed there under *non-goals* for the current phase, with a note that `polarway-distributed` exists for that later. There is no reference anywhere in the plan, or in the crate, to pysic-rs, to a physics workload, or to a compute task that is not a DataFusion query. **No `pysic-rs` ↔ `polarway-distributed` integration exists today.** What follows is a description of how one would be wired, against the APIs that exist, not a report on one that runs.

Failure mode

Treating Table 3’s partial modules as a working scheduler without opening the source would be the infrastructure analogue of Section 5’s vacuous test: a claim that cannot be falsified because no one checked what would have to be true for it to fail. The question we asked before writing this section — “could I be wrong that this dispatches work?” — is the same question Section 5 recommends asking of a numerical assertion.

Module	Status	What the source actually does
<code>coordinator.rs</code>	stub	<code>register_worker</code> , <code>get_workers</code> and <code>become_leader</code> are async functions that log a message and return <code>Ok(())</code> , <code>Ok(vec![])</code> and <code>Ok(false)</code> . Every line that would talk to etcd is commented out behind a <code>TODD</code> .
<code>query_planner.rs</code>	stub	<code>QueryPlanner::plan</code> ignores the structure of its query string and always returns exactly one <code>ExecutionStage</code> with hard-coded <code>estimated_rows: 1000</code> . <code>optimize()</code> is a pass-through that returns its argument unchanged; the doc comment lists filter push-down and shuffle minimisation as work not yet started.
<code>executor.rs</code>	partial	<code>assign_stages</code> is real: it round-robins each stage of a <code>QueryPlan</code> across the <code>WorkerInfo</code> entries with <code>available && current_load < max_load</code> . But <code>execute_stages</code> , which is supposed to run the assigned stages, logs their description and unconditionally returns <code>Ok(vec![])</code> — no network call is made and no work is executed.
<code>cache.rs</code>	working	A moka-backed cache keyed on <code>(query, params_hash, schema_version)</code> with TTL/TTI eviction, exercised by two passing unit tests. This is the one module here that does what its name says.

Table 3: `polarway-distributed`, module by module, against the plain meaning of its own doc comments. No file in this crate defines a `.proto` service or a `build.rs`, so the `tonic/prost` dependencies compile but generate nothing inside this crate.

9.2 The wiring pattern

Nothing in `polarway-distributed` refers to a compute payload; every struct is shaped around a SQL string and an Arrow `RecordBatch`. A `pysic-rs` sweep does not have a query string, but it does produce exactly the kind of columnar output `DistributedExecutor::execute` already returns: a Dirac norm-drift trace or an eigenvalue array is a one- or two-column `RecordBatch` like any other. The natural integration point is therefore not to bypass these types but to fill them with physics:

- **Planning.** Replace `QueryPlanner::plan`'s single hard-coded stage with one `ExecutionStage` per grid point of the sweep (or per contiguous block of grid points, to amortise dispatch overhead against sweeps of a few hundred points). The `estimated_rows` / `estimated_size_bytes` fields, which `plan()` currently sets to constants, would carry a real cost model — for the split-step Dirac propagator, proportional to $N_{\text{steps}} \times N_{\text{grid}}$ — so that `assign_stages`'s round-robin, which today ignores cost entirely, has something to balance against.
- **Dispatch.** `ExecutionStage.description` is a free-form `String` today (`format!("Execute query: {}", query)`); the field that would actually need to exist is a serialized job descriptor — routine name, parameters, step count — carried as bytes rather than as a human-readable sentence. Making `execute_stages` do something other than log and return `Ok(vec![])` means writing the missing piece: a gRPC service (`tonic/prost` are already dependencies, but no `.proto` defines one in this crate) with one RPC that takes the serialized job and returns a `RecordBatch`.
- **Execution, per shard.** The worker process behind that RPC is where `pysic-rs` itself runs — a small binary that deserializes the job and calls the corresponding Rust entry point directly (e.g. the Dirac split-step propagator of §4), the same code path exercised on a single core throughout this paper, just invoked once per shard instead of once per figure.
- **Caching.** `CacheLayer<V>` is the one module that needs no modification: keying on `CacheKey::new(routine_name, params_hash, schema_version)` lets a resubmitted grid point — after a dead worker, say — be served from cache instead of recomputed, exactly the use case its TTL/TTI eviction is already built for.

9.3 Where optimiz-rs fits

pysic-rs's own Cargo.toml already declares `optimizr = { version = "2.0.0", package = "optimiz-rs", default-features = false }`. We checked whether any module under `pysic-rs/src` actually calls it: none does. Every solver described in Section 4 — the split-step propagators, the tridiagonal eigensolver, the ADM and Wheeler–DeWitt integrators — is pysic-rs's own Rust, not a call into optimiz-rs. The dependency is present and currently unused.

That leaves optimiz-rs's realistic role in a distributed sweep as *orchestration*, not *solving*. optimiz-rs ships a family of CPU-only, dependency-free primitives — `grid_search` for laying out a parameter grid, `differential_evolution` and `mcmc` for sampling one — that a coordinator or a worker can call without pulling in anything pysic-rs does not already link. Laying out the $(\Delta t, m)$ grid for a Dirac sweep with `optimizr::grid_search` instead of a Python `itertools.product` loop keeps grid generation in the same language and the same process as the workers that will consume it; fitting a post-hoc model to a measured convergence curve (e.g. verifying the $\mathcal{O}(\Delta x^2)$ scaling of Figure 2 across a wider range than was measured by hand) is a natural use of `differential_evolution` once the shard results are back. Neither use requires optimiz-rs to know anything about quantum mechanics or general relativity; it supplies the search and sampling scaffolding around a sweep whose numerical content is entirely pysic-rs's.

9.4 Sketch, not shipped

The following is not code that exists in either repository. It shows the shape the two missing pieces from Table 3 — a planner that emits more than one stage, and an executor that actually dispatches — would need to take, expressed against the real field names of `QueryPlan`, `ExecutionStage` and `WorkerInfo`.

```
// SKETCH: not present in polarway-distributed today.
// A planner that emits one stage per (dt, mass) grid point instead of
// query_planner.rs's single hard-coded stage.
struct DiracSweepJob { dt: f64, mass: f64, n_steps: usize, n_grid: usize }

fn plan_dirac_sweep(grid: &[DiracSweepJob]) -> QueryPlan {
    let stages = grid.iter().enumerate().map(|(id, job)| ExecutionStage {
        id,
        // today this field is a human-readable sentence; here it carries
        // a serialized job instead (bincode/serde, not shown)
        description: serde_json::to_string(job).unwrap(),
        assigned_worker: None,
        dependencies: vec![], // grid points are independent
        estimated_rows: job.n_steps, // real cost signal, not a constant
        estimated_size_bytes: job.n_grid * 16,
    }).collect();

    QueryPlan {
        id: Uuid::new_v4(),
        query: "dirac_split_step_1d sweep".into(),
        logical_plan: format!("{}", independent_shards, grid.len()),
        stages,
        estimated_cost: grid.iter().map(|j| (j.n_steps * j.n_grid) as f64).
            sum(),
    }
}

// SKETCH: the missing half of executor.rs -- execute_stages today logs
// each stage and returns Ok(vec![]) unconditionally. Dispatch would need
// a gRPC client call per assigned stage (no .proto exists for this yet):
async fn run_shard_on_worker(worker: &WorkerInfo, job: &DiracSweepJob)
    -> Result<RecordBatch>
{
```

```

// on the worker side, this deserializes 'job' and calls straight
// into pysic-rs's own solver -- the same routine graded in Sec. 3:
// let (norm_trace, times) = pysic::pde::dirac::dirac_split_step_1d(
//     job.mass, job.dt, job.n_steps, job.n_grid);
// optimiz-rs enters only if the *sweep itself* needs a search step,
// e.g. optimizr::grid_search to lay out 'grid' in the first place.
todo!("gRPC call to {}"; no such RPC is defined in this crate", worker.
    endpoint)
}

```

Remark 9.1. Every `todo!` and unused struct field above is deliberate. The point of this section is not to claim a benchmarked pipeline — none was run, and none of the numbers in this paper were produced this way — but to show, concretely, which two modules of `polarway-distributed` would need real bodies, and which one (`cache.rs`) would not, if `pysic-rs` were ever run this way.

10 Conclusion

Rust eliminated an entire class of defects and none of the defects we actually had. Every bug reported here was a piece of mathematics transcribed incorrectly into code that the compiler had no way to question: a sine exponentiated instead of multiplied, a floor applied to a real argument, a sign inverted on a correction term.

What caught them was placing each number next to an independently known value. That discipline is inexpensive when the physics supplies the invariant, and unitarity, vacuum conditions and integrality supply a great many. Our recommendation is narrow and, we think, actionable: for each routine, write down the quantity it must preserve before writing the routine, and treat any test that cannot fail as a defect in its own right.

Reproducibility

Source, tests, notebooks and the scripts generating every figure in this paper: <https://github.com/ThotDjehuty/pysic-rs>. Documentation: <https://pysic-rs.readthedocs.io/>. The companion notebook `14_dirac_adm_wheeler_dewitt.ipynb` reproduces every measured quantity quoted here.

References

- [1] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, 1964.
- [2] Y. Aharonov and D. Bohm, “Significance of Electromagnetic Potentials in the Quantum Theory”, *Phys. Rev.* **115**, 485, 1959.
- [3] R. Arnowitt, S. Deser and C. W. Misner, “Dynamical Structure and Definition of Energy in General Relativity”, *Phys. Rev.* **116**, 1322, 1959.
- [4] M. V. Berry, “Quantal Phase Factors Accompanying Adiabatic Changes”, *Proc. R. Soc. Lond. A* **392**, 45, 1984.
- [5] B. S. DeWitt, “Quantum Theory of Gravity. I. The Canonical Theory”, *Phys. Rev.* **160**, 1113, 1967.
- [6] P. A. M. Dirac, “The Quantum Theory of the Electron”, *Proc. R. Soc. Lond. A* **117**, 610, 1928.
- [7] M. Galassi et al., *GNU Scientific Library Reference Manual*, 3rd ed., Network Theory Ltd., 2009.
- [8] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3rd ed., Cambridge University Press, 2007.

- [9] B. T. Smith et al., *Matrix Eigensystem Routines — EISPACK Guide*, Springer, 1976.
- [10] P. Virtanen et al., “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”, *Nature Methods* **17**, 261, 2020.