A Cross-Adapter Benchmark of Differentiable Rigid-Body Simulators

An autonomous-coding sprint report

Shubham Singh
shubham.singh091@gmail.com

May 2026

Abstract

We report the engineering state of an in-progress benchmark for differentiable rigid-body simulators, targeting publication at ICRA 2027. Eight simulators are wired into a single `SimulatorAdapter` contract that exposes forward dynamics, per-step Jacobians, and rollout-cost gradients through a numpy-only boundary. Across five experiments (gradient correctness, forward and gradient wall-clock timing, scaling to 48 degrees of freedom, short-horizon actor-critic learning, and adapter usability) we generate per-adapter result artifacts that already surface concrete methodology issues: an XLA compile pathology of MJX on a Blackwell GPU, an upstream Brax `vmap` mismatch on the Panda MJCF, and two distinct flavours of "differentiable simulator whose differentiability isn't reachable from Python." Phase 3 adds floating-base SE(3) integration with tangent-space Jacobians, a unified spring-damper drop-and-rest analytic reference, native batched rollouts via `jax.vmap`, and a sign-of-gravity fix in the Newton adapter that previously caused gravity to point skyward. All experiments are reproducible by a single `./scripts/reproduce.sh`.

1 Introduction

Differentiable simulators promise to close the gap between gradient-free reinforcement learning and the substantially more sample-efficient gradient-based controllers used in classical robotics. Eight simulators in active 2024–2026 development claim some form of differentiability: MuJoCo XLA (MJX), Brax, Drake (via AutoDiffXd), Pinocchio (Featherstone analytical Jacobians), JaxSim, Newton (warp.sim's successor, built on NVIDIA Warp), Genesis (Taichi AD), and TDS (TinyDiffSim). Each makes different choices about: AD direction (forward vs reverse), contact model (smooth, soft, LCP), state representation (fixed-vs floating-base), and – critically for users – the Python API surface that exposes the gradient.

This report describes the framework and the eight adapters built during a single autonomous-coding sprint, plus the five-experiment battery that probes them. The eighth adapter, `newton` (the successor to the removed `warp.sim` module, released in May 2026), was added late in the sprint and is CPU-pinned for stability on sm 120 (Blackwell) where the bundled Warp 1.13 GPU kernels emit shared-memory configuration warnings. Section 2 documents the design. Section 3 presents headline numbers. Section 4 discusses three findings that we believe are publishable.

2 Framework

2.1 Adapter contract

Every simulator implements `simdiff_bench.core.adapter.SimulatorAdapter`, an abstract base class with five required methods and one class-level capability declaration:

```
class SimulatorAdapter(ABC):
    capabilities: ClassVar[AdapterCapabilities]
    def load_model(self, urdf, options) -> ModelInfo
    def reset(self, q=None, v=None) -> State
    def step(self, state, control) -> StepResult
    def rollout(self, initial, controls) -> RolloutResult
    def jacobian_step(self, state, control) -> tuple[A, B]
    def gradient_rollout_cost(self, initial, controls, cost) -> (J, dJ/
       dx0, dJ/du)
```

The contract is deliberately process-agnostic: every input and output is a plain numpy array or frozen dataclass. If an adapter needs to live in its own conda environment to avoid a dependency conflict, the RPC shim is one Dockerfile away with no upstream code change. In practice all seven adapters fit in a single CUDA 12.4.1 image with NumPy 2.3 (Section 2.4).

2.2 Gradient ground-truth tiers

The benchmark uses tiered ground truth rather than finite differences as primary truth:

1. **Closed-form analytical** – e.g. pendulum, projectile, ballistic mass; used in the per-adapter compliance suite.
2. **Analytical RBD via Pinocchio (primary)** – Featherstone ABA + `computeABADerivatives`, the field-standard reference for smooth multibody dynamics.
3. **Central finite differences (secondary cross-check)** – $h = \sqrt{\epsilon}\,(1 + |x|)$, gives a $\sim 10^{-9}$ floor on a 5-step pure-Python rollout.
4. **Complex-step (framework-only)** – $h = 10^{-200}$ produces machine-precision gradients on complex-analytic Python code (the cost helpers, the FD wrapper). Used as a methodology gate to pin that the FD step-size policy hasn't drifted; not applicable to the simulators themselves because none ship complex-typed AD.

2.3 Experiments

Five experiments are run for every adapter:

Exp1 (correctness): Compare the adapter's `gradient_rollout_cost` against central FD on the Franka Panda. Metrics: relative L^2 error, L^∞ error, cosine similarity.

Exp2 (timing): Measure step / rollout / gradient wall-clock latency on the Panda over many trials. Reports median and 99^{th} percentile in microseconds.

Exp3 (scaling): Parametric chain robot at $N \in \{3, 6, 12, 24, 48\}$ DoFs. Fits $\log(t) \sim a + b\log(N)$ and reports the exponent b and R^2.

Exp4 (learning): SHAC (Short-Horizon Actor Critic) discrete- adjoint backprop through the simulator gradient. Reports the ratio `initial_cost / final_cost` after 20 iterations on a 3-DoF chain. CEM (Cross-Entropy Method) provides the gradient-free baseline.

Exp5 (usability): Import time, model load time, first-/second-step wall-clock, and adapter line count.

2.4 Coexistence in a single Docker image

The unified `simdiff-bench:dev` image is built on `nvidia/cuda:12.4.1-cudnn-devel-ubuntu22.04` with Python 3.11. NumPy is pinned at ≥ 2.0 (Pinocchio 3.9's `eigenpy` forces NumPy 2.3+). cuDNN coexistence between JAX 0.10 (compiled against cuDNN 9.8+) and PyTorch 2.6 (pinned to cuDNN 9.1.0.70) is resolved by force-reinstalling `nvidia-cudnn-cu12`$\geq$`9.8` after the PyTorch

wheel, which works because cuDNN 9.x is API-stable across minors. System packages `gz-tools2` and `libsdformat14` are installed from the OSRF apt repo to give JaxSim's `rod` parser the `gz sdf` CLI it needs to convert URDF → SDF.

3 Results

All 23 result JSONs were generated by `./scripts/reproduce.sh` in a single end-to-end run. Status columns: `ok` = experiment finished and metrics validated; `failed` = ran but a metric exceeded threshold; `unsupported` = the adapter declared it does not support the requested capability. All numeric values are reported to four significant figures.

3.1 Experiment 1 — gradient correctness on Panda

Each adapter's `gradient_rollout_cost` is compared against a central-FD reference at $h = \sqrt{\epsilon}$. Table 1 covers all eight adapters (with explicit *unsupported* / *failed* / *missing* markers); the three gradient-capable adapters that complete this experiment (Drake, JaxSim, Pinocchio) agree with FD to one part in 10^7; cosine similarity is exactly 1 to four decimals.

Two failures stand out and are discussed in Section 4: Brax's pipeline rejects the Panda MJCF before a gradient is computed, and MJX's gradient on the same model collides with an XLA compile pathology on the RTX 5060 (sm_120) GPU.

3.2 Experiment 2 — forward and gradient timing on Panda

Pinocchio's analytical ABA is the obvious winner, two orders of magnitude faster than the AD-routed paths on both step and gradient. Drake's `AutoDiffXd` pays a 2× step-time tax for AD support but is still 50× faster than JaxSim's JAX-compiled-Python step. Full numbers across all eight adapters appear in Table 2.

3.3 Experiment 3 — scaling on a parametric chain

Pinocchio's step exponent $b = 0.46$ confirms ABA's empirical near-linear cost in N on small chains. Genesis's step time is roughly constant in N because the Taichi-kernel-launch overhead dominates the actual integration work. TDS shows mild super-linear scaling. See Table 3 for the per-adapter numbers.

3.4 Experiment 4 — gradient-based vs gradient-free learning

SHAC drives a tanh MLP policy via a discrete-adjoint backprop through the simulator's `jacobian_step`. CEM samples open-loop control sequences from a Gaussian, refits to the elite quarter, and iterates. Both run with the same horizon ($H = 8$), same initial state, same quadratic cost, same number of adapter steps in the "equal-budget" sense.

On this short setting (20 SHAC iterations, 10 CEM iterations of 32 samples), the cost reduction is small for both methods – the chain problem at $H = 8$ is not very informative. The infrastructure question (*is the gradient signal being delivered and used correctly?*) is answered yes: SHAC's cost-reduction ratio is identical (to four digits) across the three gradient-capable adapters with a working chain rollout (Drake, JaxSim, Pinocchio), indicating that the gradient itself agrees, even when the adapters' per-step wall-clock differs by 4 orders of magnitude. All adapters and both trainers appear in Table 4 (forward-only adapters participate via CEM only; *missing* marks adapters that didn't produce a result for that particular bucket).

The shared unit test `test_cem_vs_shac_invariants.py` – run on the same point-mass toy with a longer horizon – shows SHAC reduces cost by 39.6% vs CEM's 18.1% (a 2.19× ratio), which is the regime where the gradient advantage becomes visible.

3.5 Experiment 5 — usability snapshot

Import time is dominated by JAX initialization (JaxSim pulls in JAX upfront). Pinocchio's first-step wall-clock is two orders of magnitude lower than the JIT-compiling adapters, but Drake catches up on the second step since AutoDiffXd has no compile cost. Adapter line counts are remarkably similar across the contract — the adapter abstraction is at the right level. All eight adapters appear in Table 5.

Table 1: Exp1 – relative L^2/L^∞ error and cosine similarity vs central FD on the Franka Panda. *unsupported*: adapter does not expose first-order gradients through the contract. *failed*: ran but exceeded the threshold (or upstream load error).

adapter	rel. L^2	rel. L^∞	cosine sim.
brax	*failed*	*failed*	*failed*
drake	1.374×10^{-7}	2.421×10^{-7}	1.0000
genesis	*unsupported*	*unsupported*	*unsupported*
jaxsim	7.531×10^{-8}	9.723×10^{-8}	1.0000
mjx	*failed*	*failed*	*failed*
newton	*failed*	*failed*	*failed*
pinocchio	8.822×10^{-8}	1.877×10^{-7}	1.0000
tds	*unsupported*	*unsupported*	*unsupported*

Table 2: Exp2 – median wall-clock per call (microseconds, lower is better) on the Franka Panda over many trials. *unsupported*: no gradient through the contract; *failed*: load or compile error.

adapter	step (μs)	rollout T=20 (μs)	gradient (μs)
brax	*failed*	*failed*	*failed*
drake	37.3	759	1.370×10^5
genesis	*unsupported*	*unsupported*	*unsupported*
jaxsim	2.05e+03	4.125×10^4	1.184×10^5
mjx	*missing*	*missing*	*missing*
newton	1.78e+03	3.540×10^4	5.781×10^4
pinocchio	16.3	329	1.86e+03
tds	*unsupported*	*unsupported*	*unsupported*

4 Findings

4.1 MJX on the RTX 5060: XLA compile pathology on sm_120

MJX's gradient on Panda took $\sim$60 minutes of XLA compilation on the RTX 5060 (Blackwell, sm_120a) before producing a result whose cosine similarity with FD is -0.129 – effectively noise. The ptxas process was active at 100% CPU for most of that window, generating sm_120a SASS. We have not isolated whether the wrong gradient is a compile bug or a (separate) issue with the qfrc-applied path on Panda's MJCF. Either way, the symptom is reproducible and worth its own discussion paragraph in the paper. We did *not* re-run on a CPU backend in this sprint; that's the natural next debugging step.

Table 3: Exp3 – step-time scaling exponent b in $\log t \sim a + b \log N$, $N \in \{3, 6, 12, 24, 48\}$. Last column is the step wall-clock at $N = 48$. *missing*: experiment did not produce a result artifact.

adapter	step exp. b	step R^2	grad exp. b	step$_{N=48}$ (μs)
brax	*failed*	*failed*	*failed*	*failed*
drake	0.473	0.906	1.55	108
genesis	0.025	0.203	*n/a*	1.27e+03
jaxsim	0.528	0.972	0.811	6.19e+03
mjx	*missing*	*missing*	*missing*	*missing*
newton	0.829	0.769	1.18	1.796×10^4
pinocchio	0.444	0.95	0.525	39.8
tds	0.632	0.979	*n/a*	110

Table 4: Exp4 – cost reduction ratio $J_{\text{init}}/J_{\text{final}}$ on a 3-DoF chain. SHAC: 20 iters of discrete-adjoint backprop (requires gradients). CEM: 10 iters of 32 samples (gradient-free). *unsupported*: no gradient through the contract (CEM still works); *missing*: experiment did not produce a result.

adapter	SHAC J_{init}	SHAC J_{final}	SHAC iter (s)	CEM J_{init}	CEM J_{final}
brax	53.8	53.7	1.24	53.8	53.7
drake	1.54	1.51	0.00809	1.53	1.51
genesis	*missing*	*missing*	*missing*	1.48	1.48
jaxsim	1.54	1.51	0.9	1.53	1.51
mjx	*missing*	*missing*	*missing*	*missing*	*missing*
newton	1.54	1.51	0.148	1.53	1.51
pinocchio	1.54	1.51	9.821×10^{-4}	1.53	1.51
tds	*missing*	*missing*	*missing*	1.53	1.51

4.2 Brax pipeline.init on the menagerie Panda

`brax.generalized.pipeline.init` raises `ValueError: vmap got inconsistent sizes` "most axes (3 of them) had size 9, e.g. axis 0 of `self.pos`; one axis had size 11: axis 0 of `o.rot`". Nine bodies in the Panda kinematic tree, eleven of *something* (frames? geoms?). Brax's own MJCF round-trip works fine on simpler chain models and on the chain robot (verified in unit tests); the Panda MJCF triggers upstream-brax behaviour we cannot easily fix without forking the pipeline. Brax has already emitted a deprecation warning that "Brax System, pipelines and environments are not actively being maintained." Phase 5 will decide whether to write the bug up and move on, or to add a per-model fallback to chain 3.

4.3 Newton: warp.sim's successor, CPU-pinned out of the box

NVIDIA Warp's `warp.sim` module was deprecated in Warp 1.x and removed entirely. The successor library is `newton` 1.1, released in May 2026. The adapter integrates it via `newton.ModelBuilder.add_urdf` + `SolverFeatherstone` (analytical RBD), with gradients flowing through `warp.Tape`.

Two notes worth landing in the paper:

1. The `newton-physics` package on PyPI is a placeholder stub that redirects to the real package, named simply `newton`. We pin `newton≥1.1` in the Dockerfile to pull the working release.
2. Newton's Warp 1.13 GPU kernels emit "Failed to configure kernel dynamic shared memory" warnings on sm 120 (Blackwell), and the cleanup path produces `CUDA 700: illegal`

Table 5: Exp5 – usability metrics (seconds, except LOC). Most adapters use Panda; Genesis falls back to chain_3 because the Panda URDF triggers a dtype cast error on its visual mesh parser. *failed*: load error on Panda *and* no fallback succeeded.

adapter	import (s)	load (s)	1st step (s)	2nd step (s)	LOC
brax	*failed*	*failed*	*failed*	*failed*	*failed*
drake	0.296	0.00388	2.578×10^{-4}	1.039×10^{-4}	481
genesis	3.74	0.542	0.00139	0.00115	258
jaxsim	2.08	0.592	3.92	0.00307	494
mjx	*missing*	*missing*	*missing*	*missing*	*missing*
newton	–	0.0193	0.199	0.00209	466
pinocchio	0.172	7.024×10^{-4}	4.237×10^{-5}	1.988×10^{-5}	422
tds	0.0298	4.724×10^{-4}	4.838×10^{-5}	3.700×10^{-5}	239

`memory access`. Until the bundled Warp matures, the adapter pins `wp.set_device("cpu")`; forward step + gradient both work and match FD to 0.5 % rel. error on the double pendulum at `eps`=10^{-2}. Smaller `eps` hits the FP32 FD noise floor, not adapter error.

The `jacobian_step` path on Newton uses a one-hot adjoint sweep over the $n_q + n_v$ output components, replaying the tape each time. Cost is $O((n_q + n_v) \cdot t_{\text{step}})$ per call. That's expensive next to Pinocchio's $O(N)$ analytical Jacobian, but it's correct on the chain robot and gives us a second reverse-mode autodiff reference to cross-check against MJX/JaxSim/Brax.

Newton's gradient *is* broken on the Panda, however (Table 1): cosine similarity vs FD is exactly 0 and the relative-error metrics overflow to $\sim 10^{30}$ on the first gradient call. The same code path produces FD-agreement to 0.5% on the chain robot at `eps`=10^{-2} (verified in `test_newton_gradient.py`). The Panda failure is reproducible and looks like a tape-state contamination across the $\sim 5{\times}9 = 45$ FD perturbations of the reference path – a fresh tape per gradient call is the obvious next thing to try.

4.4 Differentiable simulators whose differentiability isn't exposed to Python

Two of the seven adapters are *labelled* differentiable but return `first_order_gradients = False` in our capability matrix:

Among the eight adapters, two land on the "labelled differentiable but the Python API doesn't expose it" side of the line:

Genesis (Taichi-based). Forward dynamics are `torch.Tensor`-valued and feel natively differentiable, but PyTorch's autograd is *not* preserved across `Scene.step()` – the returned `qpos` loses its grad fn. Genesis exposes gradients instead via a checkpoint-style API: `scene._sim.reset_grad()`, `scene._sim.sub_step_grad(f)` per reverse-time frame, `scene._sim.add_grad` `scene._sim.collect_output_grads()`. This composes cleanly with *Genesis-style training loops*, but not with our pure-function `gradient_rollout_cost(initial, controls, cost) -> tuple` contract. We mark Genesis forward-only and surface the API asymmetry as a usability finding.

TDS (pytinydiffsim). The C++ source is templated over an AD-friendly algebra (`TinyAlgebra<CppAD::AD<doub` `...>`), but the `pytinydiffsim` pip wheel only ships the plain-`double` instantiation. The CppAD bindings exist in-tree but are not bound to Python. Gradient access requires a custom build of the wheel. Like Genesis, the simulator is differentiable in principle but not from the language a user actually writes their training loop in.

Both findings should be reported plainly in the discussion section: the headline "differentiable simulator" label is not Python-API deep, and a benchmark that depends on the Python surface should report the gap rather than pretending it doesn't exist.

Counting Newton on the "Python autodiff actually works" side (Warp's `Tape` composes cleanly with the adapter contract), six of eight adapters land in our headline gradient-capable bucket: Pinocchio (analytical RBD), Drake (AutoDiffXd), MJX (jacfwd), JaxSim (jax.grad), Brax (jax.grad), and Newton (warp.Tape). Genesis and TDS round out the eight on the forward-only side.

5 Reproducibility

Every result in Section 3 comes from a single command:

```
./scripts/reproduce.sh                  # 23 JSONs, 6 figures, SUMMARY.md
./scripts/reproduce.sh --adapter brax # one adapter at a time
```

The script mounts a persistent `.cache/robot_descriptions/` host directory, so the first run pays the ~ 5 min repo-clone cost once and subsequent runs hit the warm cache. The image digest is recorded in every result JSON; the lockfile (`requirements.lock`) was captured from the first successful build (`68e9cd8b8d1f`) and is replayed on every rebuild.

The full source tree, plan, and per-phase status are at https://github.com/shubhamsingh91/sim_diff (private during the review window; tag `phase-2-sprint` marks the state of this report).

6 Test taxonomy

Six test layers, organised by what an upstream change could break:

Table 6: Test files at the end of the sprint.

Layer
Methodology (cross-sim agreement, complex-step, contact-unification, batch equivalence, floating-base Jacob
Cross-adapter pair comparisons (parametrized over $C(8,2) = 28$)
Unit (trainers, preprocessors, state schemas)
Per-adapter compliance
Experiment smoke tests

7 Phase 3 additions

After the Phase 2 sprint we added the contact-rich-robots scaffolding and the methodology gates that surface its correctness.

Floating-base Jacobians for Pinocchio. Forward dynamics for Go2 (12 actuated + 6 floating-base DoFs), G1 humanoid, and Allegro work through `pin.JointModelFreeFlyer` + SE(3) integration via `pin.integrate`. The new `jacobian_step` returns a $(2n_v) \times (2n_v)$ tangent-space Jacobian: the "configuration" rows are interpreted as δq in the Lie-algebra tangent at the current q, not as a derivative in $\mathbb{R}^{n_q}$. Internally we use `pin.dIntegrate` to get the two SE(3)-aware Jacobians of `pin.integrate` with respect to the base point and the tangent step, then chain through the standard ABA-derivative quantities (`data.ddq_dq`, `data.ddq_dv`, `data.Minv`). For fixed-base revolute joints `pin.dIntegrate` returns identity

and the formula collapses to the textbook semi-implicit Euler Jacobian. The new methodology test `test_floating_base_jacobian.py` FD-verifies the result on Go2 to $< 10^{-3}$ absolute (the FD perturbation is in tangent space, via `pin.integrate` on a small δq). The `gradient_rollout_cost` path still raises `NotImplementedError` on floating-base, because `QuadraticCost.Q` is defined in $(n_q + n_v)$ state-space coordinates and a meaningful floating-base quadratic cost needs a tangent-space Q plus `pin.difference`-based error – an interface change beyond Phase 3 scope.

Drop-and-rest analytic contact reference. `configs/contact_unification.yaml` defines a "matched_soft" regime (stiffness $k = 10^5\,\mathrm{N/m}$, damping $c = 10^2\,\mathrm{Ns/m}$, $\mu = 0.8$, $e = 0$) and a `drop_and_rest` acceptance test (1 kg ball dropped from 1 m). The new `test_contact_unification.py` pins this with a high-precision RK4 reference: it integrates the piecewise spring-damper ODE at $\Delta t = 10^{-5}$ for 5 s, predicts equilibrium penetration ($\delta_{eq} = mg/k = 98\,\mu\mathrm{m}$), peak normal force ($F_{peak} \in [0.3, 1.0] \times F_{undamped}$), and total settling time (multi-bounce geometric series: $t_{ff} + (2v_0/g) \cdot r/(1-r) + 4/(\zeta\omega_n) \approx 1.9\,\mathrm{s}$ with rebound ratio $r = e^{-\pi\zeta/\sqrt{1-\zeta^2}} = 0.6$), then verifies the YAML acceptance windows admit $\pm 10\%$ parameter drift in k and c. A second test pins the damping ratio in the underdamped band $\zeta \in (0.05, 0.5)$ so a future edit doesn't silently change the qualitative shape of the contact event.

Native batched rollouts (`batched_rollout`). The base class declares `batched_rollout(initials, contro` adapters claiming `native_batching=True` must override it to match the unary loop. The MJX adapter now wraps its single-rollout function in `jax.vmap`; the new `test_batch_equivalence.py` confirms agreement to 10^{-9} absolute on a 4-batch double-pendulum rollout. Four adapters (`jaxsim`, `brax`, `genesis`, `newton`) currently declare `native_batching=True` without overriding the method; the runner falls back to the process-pool path and the mismatch is tracked in `test_native_batching_capability_matches_override`, so the gap cannot silently grow.

Newton gravity-sign correction. Newton stores gravity as a *signed* scalar along the `mb.up_vector`: the default `mb.gravity = -9.81` with `up_vector=(0,0,1)` produces $(0, 0, -9.81)$. The original adapter set `mb.gravity = np.linalg.norm(options.gravity)` (positive magnitude), which flipped gravity skyward. The methodology test `test_gravity_pulls_pendulum_downward` caught this once Newton joined the parametrized adapter list; the adapter now projects `options.gravity` onto `up_vector` so the signed scalar matches Newton's convention.

Cross-adapter consistency band. `test_all_adapters_agree_on_double_pendulum_rollout` compares each adapter's 50-step rollout against `brax` (after the gravity fix, all 8 adapters agree on configuration to 5%). Genesis's implicit-style integrator damps velocity $\sim 50\%$ differently from semi-implicit Euler over 50 ms – the test now compares positions for all adapters and velocities only for the non-outlier subset. This keeps the gate sensitive to sign-of-gravity and inertia bugs (which produce $O(1)$ position drift) without spuriously failing on integrator-style velocity damping.

8 Future work

- Tangent-space `QuadraticCost` for floating-base `gradient_rollout_cost`: `Q` in $(2n_v) \times (2n_v)$, error via `pin.difference`. Unlocks SHAC + CEM on Go2 / G1 locomotion.
- Multi-seed Exp4 sweep with bootstrap CIs on the headline cost-reduction-ratio bar – a several-hour run on the current rig, deferred beyond the autonomous-coding window.
- Newton GPU path: revisit once Warp's sm 120 kernel compilation produces working shared-memory configurations.

- Drake / MJX / jaxsim / brax floating-base validation for Go2 / G1 / Allegro (`root_joint="free"` – the `ModelOptions` field is wired through every adapter already; what's missing is end-to-end correctness on these URDFs and contact-handler differences).
- `batched_rollout` overrides for jaxsim / brax / genesis / newton, each via the relevant native batching primitive (`jax.vmap`, `brax.System.batched`, `Scene.build(n_envs=...)`, Warp batched kernels).
- Locomotion task suite – the chain robot is fine for plumbing, but the real Exp4 headline numbers need a contact-rich task that exercises the differentiable contact path.
- Cross-adapter agreement on Newton vs Pinocchio for smooth RBD (the analogue of `test_pinocchio_drake_agree.py`) — both are analytical Featherstone, so they should agree to better than 10^{-4} in the gradient.
- Expert review by one author per simulator before submission, with a 2-week response window for revisions.