Analytical Rigid-Body Second-Order Derivatives vs. CasADi Autodiff for Whole-Body MPC: A Python-Binding Benchmark

Shubham Singh

May 2026 | Technical note

Abstract

Model-predictive control (MPC) of legged robots spends most of its per-tick budget computing derivatives of the rigid-body dynamics. Two routes dominate practice: symbolic auto-differentiation with code generation (CasADi), or closed-form *analytical* rigid-body-dynamics (RBD) derivative algorithms. Comparisons between them are notoriously easy to get wrong. This note gives a controlled, apples-to-apples benchmark of the *second-order* inverse-dynamics derivatives on a Unitree B2+Z1 (quadruped with a 6-DoF arm; n_q=26, n_v=25, free-flyer), measured through *identical Python bindings* in the controller's own deployment environment. Holding the computed quantity, the abstraction level, and the environment fixed — and explicitly avoiding the measurement traps that distort such studies — the analytical algorithm matches or beats compiled CasADi (**1.9**× first-order, **1.2**× second-order) while eliminating the symbolic code-generation wall entirely: **0** vs **8.83 MB** of generated C, **63 s** of `gcc -O2`, and **1.40 GB** of peak compiler memory for a single second-order function.

1 Setting

A whole-body MPC [1] tick solves a nonlinear optimal-control problem (OCP) over a short horizon; the solver (here Fatrop) repeatedly evaluates the constraint Jacobian (first-order dynamics derivatives) and the Lagrangian Hessian (second-order, costate-contracted). On the B2+Z1 with a 14-node OCP we measure the full loop solving at **58.8 ms/tick** (uncompiled), of which ∼80% is derivative evaluation — the Lagrangian Hessian alone is ∼45%. Making those derivatives cheaper is therefore the highest-leverage target.

Prior work shows analytical second-order RBD derivative algorithms can beat autodiff in C++ [4, 5]. The practical question for a Python-based controller stack is whether that advantage *survives at the Python-binding level* — i.e. once both algorithms are reached the way the codebase actually calls them. We answer it for the inverse-dynamics (RNEA) second-order derivatives.

2 What is compared (the same quantity)

With $\tau = \mathrm{RNEA}(q, v, a)$ the inverse-dynamics torque, the complete set of nonzero second-order derivatives is the four third-order tensors

$$\frac{\partial^2 \tau}{\partial q\, \partial q}, \quad \frac{\partial^2 \tau}{\partial v\, \partial v}, \quad \frac{\partial^2 \tau}{\partial q\, \partial v}, \quad \frac{\partial^2 \tau}{\partial a\, \partial q},$$

each $n_v \times n_v \times n_v$ ($\partial^2\tau/\partial a^2 = \partial^2\tau/\partial v \partial a = 0$, since τ is affine in a). *Both* sides compute exactly these:

- **Analytical:** Pinocchio [2] `ComputeRNEASecondOrderDerivatives` (a fork that returns the four tensors to Python).
- **CasADi:** an `SX` RNEA built exactly as the MPC builds its dynamics ($\tau = \mathrm{rnea}(\mathrm{integrate}(q, \delta q), v, a)$), with the same four blocks formed by `jacobian`-of-`jacobian`, q, v, a as symbolic inputs; code-generated, compiled with `gcc -O2`, and loaded through `ca.external` [3].

3 Methodology (and the traps avoided)

Both algorithms are compiled C reached through *one thin, equal* Python crossing (a Pinocchio binding call vs `ca.external`); the input/output marshalling tax is symmetric and cancels, so the measurement isolates algorithm vs algorithm. The recurring mistakes, each of which manufactures a wrong conclusion, are explicitly excluded:

- **No `ca.Callback`.** Wrapping the analytical code in a CasADi callback puts Python *inside* the solver's eval loop and adds ∼420 μs/call of boundary overhead — an integration artifact that swamps the ∼25 μs algorithm and falsely reverses the result.
- **Compiled, not interpreted.** The CasADi side is code-generated and `gcc -O2` compiled; timing the interpreted `SX` graph is ∼10× artificially slow.

- **Same quantity.** Full tensors are compared against full tensors — not against the λ-contracted Hessian (which is $\sim$25$\times$ less work).
- **Same environment.** Everything runs in the controller's own `conda` env.

Validation. Before timing, the two implementations are checked to compute the identical object: they agree to $\leq 10^{-15}$ (max relative error) on the joint blocks of all four tensors. The single expected discrepancy is the base-orientation block of $\partial^2\tau/\partial q^2$, which differs by the $SO(3)$ connection term (the geometrically-correct intrinsic Hessian vs the chart Hessian taken through `integrate`) — not an error.

4 Results

All numbers are for the B2+Z1 ($n_v = 25$), measured in the deployment env.

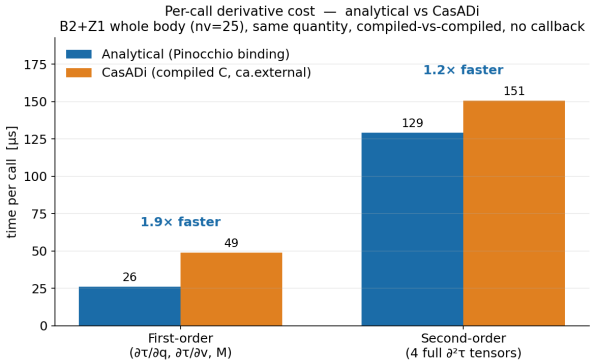

Figure 1: Per-call derivative cost (μs/call). Same quantity, compiled-vs-compiled, no callback.

Table 1: Memory / build footprint of the second-order derivatives.

axis (second-order)	analytical	CasADi (C)
generated artifact	**0** (in libpinocchio)	8.83 MB C $\rightarrow$ 2.96 MB `.so`
peak build RAM (`gcc -O2`)	**0**	1.40 GB (OOM@32 GB heavier)
runtime scratch / eval	`Data` $\sim$1.4 MB (shared)	14 KB work vector
output tensors	488 KB dense	83 KB sparse (10,656 nnz)
instruction count	—	393,024

5 Takeaways

(1) Raw speed reproduces the analytical advantage at the binding level. Analytical matches or beats compiled CasADi on the identical object (1.9$\times$ first-order, 1.2$\times$ second-order). The second-order margin is modest at $n_v = 25$ and widens with model size; the often-cited "analytical is slower" result is an artifact of the `ca.Callback` integration path, not the algorithm. **(2) The decisive, robust win is the code-generation wall.** Analytical pays *no* generated code, *no* compile time, and *no* compiler memory, and is fixed for any model — whereas CasADi needs a freshly generated and compiled function per object, whose build cost grows with n_v until the compiler runs out of memory. Models that CasADi cannot build at all become buildable. **(3) Realizing the online speedup** inside the deployed solver requires a compiled (non-`Callback`) integration of the analytical derivatives; the Python-callback shortcut is a trap. Code: github.com/shubhamsingh91/pinocchio.

References

[1] L. Molnar, J. Cheng, G. Fadini, D. Kang, F. Zargarbashi, S. Coros, "Whole-body inverse-dynamics MPC for legged loco-manipulation," *IEEE Robotics and Automation Letters*, vol. 11, no. 1, pp. 898–905, 2026. Code: `https://github.com/lukasmolnar/wb-mpc-locoman`.

[2] J. Carpentier *et al.*, "The Pinocchio C++ library," *IEEE/SII*, 2019.

[3] J. A. E. Andersson *et al.*, "CasADi – a software framework for nonlinear optimization and optimal control," *Math. Prog. Comp.*, 2019.

[4] S. Singh, R. P. Russell, P. M. Wensing, "Efficient analytical derivatives of rigid-body dynamics using spatial vector algebra," *IEEE RA-L*, 2022. arXiv:2105.05102.

[5] S. Singh, R. P. Russell, P. M. Wensing, "Second-order analytical derivatives of rigid-body dynamics with multiple-shooting DDP," 2023. arXiv:2307.12606.