`scipy.spatial.transform`: Differentiable Framework-Agnostic 3D Transformations in Python

Martin Schuck
Learning Systems and Robotics Lab
Technical University of Munich

Alexander von Rohr
Learning Systems and Robotics Lab
Technical University of Munich

Angela P. Schoellig
Learning Systems and Robotics Lab
Technical University of Munich

Abstract

Three-dimensional rigid-body transforms, i.e. rotations and translations, are central to modern differentiable machine learning pipelines in robotics, vision, and simulation. However, numerically robust and mathematically correct implementations, particularly on $\mathrm{SO}(3)$, are error-prone due to issues such as axis conventions, normalizations, composition consistency and subtle errors that only appear in edge cases. SciPy's `spatial.transform` module is a rigorously tested Python implementation. However, it historically only supported NumPy, limiting adoption in GPU-accelerated and autodiff-based workflows. We present a complete overhaul of SciPy's `spatial.transform` functionality that makes it compatible with any array library implementing the Python array API, including JAX, PyTorch, and CuPy. The revised implementation preserves the established SciPy interface while enabling GPU/TPU execution, JIT compilation, vectorized batching, and differentiation via native autodiff of the chosen backend. We demonstrate how this foundation supports differentiable scientific computing through two case studies: (i) scalability of 3D transforms and rotations and (ii) a JAX drone simulation that leverages SciPy's `Rotation` for accurate integration of rotational dynamics. Our contributions have been merged into SciPy main and will ship in the next release, providing a framework-agnostic, production-grade basis for 3D spatial math in differentiable systems and ML.

1 Introduction

Rigid-body rotations and translations underpin modern learning systems in robotics, vision, graphics, state estimation, and control. Their correct use requires precise numerics on Lie groups and algebras, in particular $\mathrm{SO}(3)$ and $\mathrm{SE}(3)$, to ensure stability of composition, interpolation, and differentiation. Foundational treatments with emphasis on both the geometric structure and numerically stable parameterizations are well established in mathematics, robotics, and estimation texts [1, 3, 4, 5].

In practice, practitioners either reimplement 3D transform routines to integrate with autodiff and accelerators, which can lead to subtle bugs around singularities, small angles, and representation conventions, or use framework-specific packages. JAX [2] offers a native SciPy [13] reimplementation, which is planning to drop support for 3D transformations in a future release [1], while PyTorch's [7] ecosystem relies on third-party packages such as `Kornia` [10] and `PyTorch3D` [9] to provide

[1] See *JAX Enhancement Proposal (JEP) 18137* at `https://docs.jax.dev/en/latest/jep/18137-numpy-scipy-scope.html#scipy-spatial`

1st Workshop on Differentiable Systems and Scientific Machine Learning @ EurIPS 2025

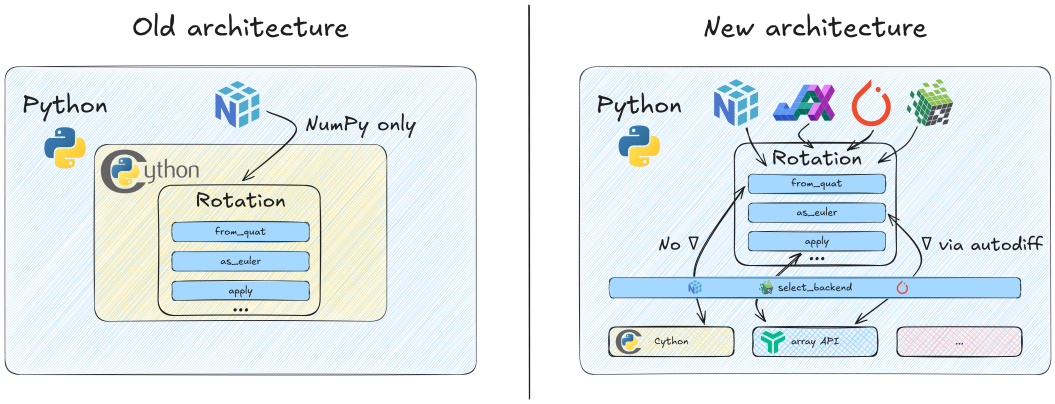

Figure 1: Overview over the changed architecture using `Rotation` as example. Instead of a pure Cython implementation which is only compatible with NumPy, the classes are now pure Python. Each method delegates computations to a backend that is selected depending on the array type. NumPy arrays are passed to the specialized Cython backend, whereas other array API frameworks use the new, generic backend. This enables advanced capabilities such as differentiation through transforms.

geometry utilities. The fragmentation into framework-specific packages has so far prevented the development of a unified, portable solution. SciPy's `spatial.transform` module has provided rigorously tested transformation algorithms in 3D, yet it historically targeted NumPy only and supported at most scalar or 1D stacks of rotations, precluding general broadcasting, autodiff, and accelerator execution.

Orthogonally, SciPy is adopting the Python array API [8], a standardized subset of NumPy-like semantics defined by the Python Data API Consortium[2]. The array API enables library authors to write backend-agnostic numerical code that runs, unmodified, on compliant array implementations such as NumPy, JAX, PyTorch, and CuPy [6], thereby inheriting those backends' accelerators, JIT compilation, and autodiff.

This paper outlines the overhaul of SciPy's `spatial.transform` module to conform to the array API and showcases how the new capabilities can be leveraged for differentiable simulation and ML. The revised implementation is framework-agnostic: it executes on CPUs and GPUs/TPUs, is compatible with native autodiff and JIT in the selected backend, and offers numerically robust $\mathrm{SO}(3)$ and $\mathrm{SE}(3)$ primitives with consistent behavior across libraries. Beyond prior functionality, we add efficient broadcasting across arbitrary leading dimensions, enabling workloads with multiple batch axes common in machine learning. Together, these changes provide a stable, rigorously tested foundation for differentiable 3D spatial computations without reimplementation burden. The overhaul has been merged into SciPy's main branch and will be included in the next release, aligning with the broader effort to reduce parallel reimplementations (e.g., in `jax.scipy`) as SciPy becomes fully array API compatible.

2 Framework-Agnostic Rotations and Translations

Previous Design and Limitations The original `spatial.transform` implementation was tightly coupled to NumPy and entirely written in Cython for improved performance. It assumed CPU execution, offered limited broadcasting (only scalars or single-axis stacks), and was incompatible with JIT compilation or automatic differentiation. As a result, downstream users reimplemented core $\mathrm{SO}(3)$/$\mathrm{SE}(3)$ functionality per framework, duplicating effort and risking subtle numerical errors.

Revised Architecture The Cython implementation offers speed advantages over a native Python implementation with NumPy, especially for common use cases such as single rotations or transforms. However, it cannot support arbitrary array API frameworks. To maintain the speedups for NumPy while still supporting other frameworks, we redesigned the architecture of the `spatial.transform` module. The established object-oriented interface is preserved, but methods now delegate to functional

[2] `https://data-apis.org/array-api`

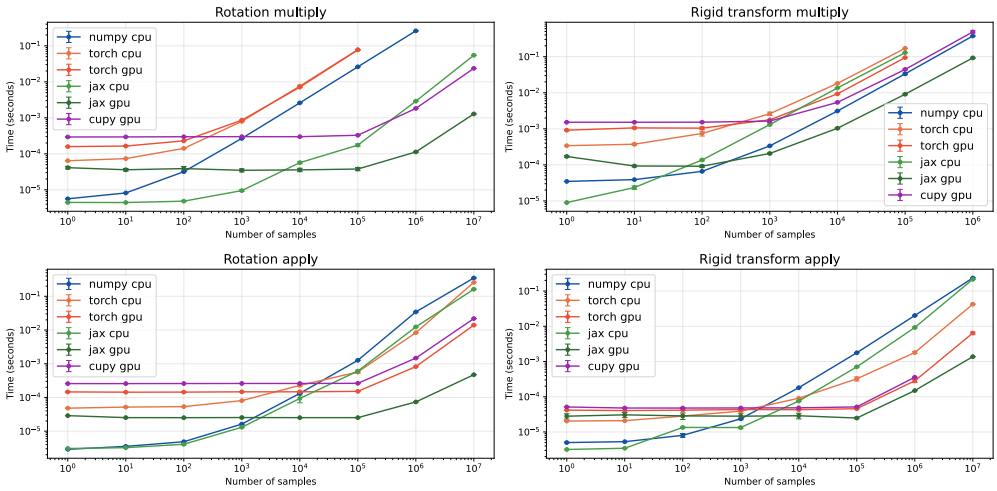

Figure 2: Computation time versus number of samples N for the multiply and apply operations of `Rotation` and `RigidTransform` across array API backends. JAX timings are evaluated after JIT compilation. GPU backends incur a fixed overhead at small N but achieve higher asymptotic throughput for large N. Notably, JAX often performs on par or better than the custom Cython backend for numpy.

backends selected by the array namespace of their inputs. Fig. 1 provides an overview over the revised architecture. NumPy arrays are processed via the Cython backend, whereas other frameworks use a new reimplementation of rotations and rigid transforms that is fully compatible with the Python array API and thus works for inputs from any frameworks. Each functionality has been rewritten in framework-agnostic, non-branching code, enabling execution on CPUs, GPUs, and TPUs; compatibility with native autodiff and JIT of the chosen framework; and consistent semantics for dtypes, devices, and promotion rules.

Data Model and Backends Rotations are represented canonically as unit quaternions, rigid transforms as a homogeneous transformation matrix. All constructors and operations support broadcasting across arbitrary leading dimensions, allowing multiple batch axes common in ML workloads without specialized code paths. Third-party arrays that implement the array API are supported out of the box. Advanced users can register custom backends to override operations with optimized versions that are not possible under the limitations of the array API while continuing to use the SciPy interface. These backend are then automatically selected at runtime depending on the array type. This design provides a single, rigorously tested foundation for accurate $\mathrm{SO}(3)$/$\mathrm{SE}(3)$ computations across frameworks, eliminating the need for reimplementations.

3 Case Studies

In this section, we showcase the capabilities of the new framework-agnostic implementations. First, we evaluate the performance of different frameworks on standard tasks such as rotations and transforms. Then, we demonstrate how the new `Rotation` class can be used to yield correct analytical gradients for the rotational dynamics using a newly developed differentiable drone simulator written in JAX.

3.1 Performance per Framework

We measure throughput for two core operations in the new implementation as a function of the number of rotations or transforms N. The microbenchmarks cover composition and application to 3D points. We evaluate NumPy, PyTorch (CPU and CUDA), JAX (CPU and CUDA), and CuPy. The results can be seen in Fig. 2. For small N, CPU backends are faster due to lower overhead timings. GPU backends underperform at small batch sizes, then improve as N increases. JAX with JIT compilation attains the highest throughput across the tested workloads; the compiler eliminates array

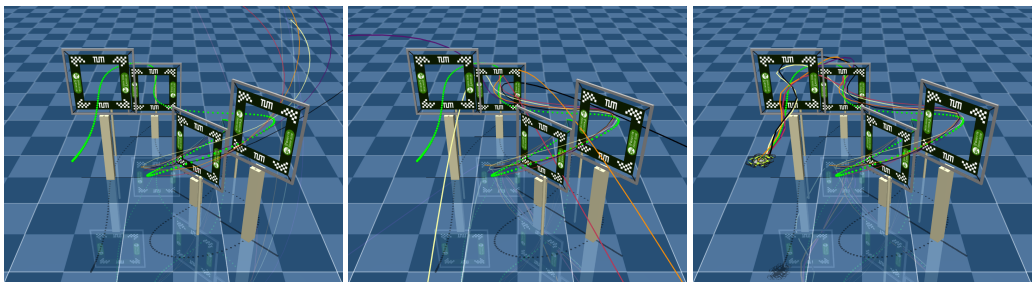

Figure 3: Several drone trajectories optimized with fully differentiable dynamics and drone controllers leveraging `scipy.spatial.transform.Rotation`. Optimization progress from left to right, reference positions in green. The visualization is based on MuJoCo [12].

API dispatch overhead and fuses kernels. For large N, all GPU backends scale favorably and surpass their CPU counterparts. The crossover depends on the operation and hardware but consistently occurs at sufficiently large batch sizes. NumPy with the Cython-backed SciPy path remains competitive for small N and single transforms, reflecting the optimization targets of the legacy implementation.

3.2 Differentiable Drone Simulation

We integrate the new Rotation primitives into a quadrotor simulator to model rigid-body rotational dynamics and differentiate through the simulation. Let $\boldsymbol{q}$ be the quaternion describing the current attitude of the drone, $\boldsymbol{\omega}_b \in \mathbb{R}^3$ the angular velocity in the body frame, $\boldsymbol{J}$ the inertia matrix and $\boldsymbol{\tau}_b \in \mathbb{R}^3$ the applied control torques in the body frame. The rotational dynamics are then

$$\dot{\boldsymbol{\omega}}_b = \boldsymbol{J}^{-1}\left(\boldsymbol{\tau}_b - \boldsymbol{\omega}_b \times (\boldsymbol{J}\boldsymbol{\omega}_b)\right), \qquad \dot{q} = \tfrac{1}{2}(\mathbf{q} \otimes \boldsymbol{q}_{\boldsymbol{\omega}_b}), \tag{1}$$

with the pure-$\boldsymbol{\omega}$ quaternion $\boldsymbol{q}_{\boldsymbol{\omega}_b} = [\omega_x, \omega_y, \omega_z, 0]$. We integrate these dynamics using the group exponential, which preserves unit quaternions by construction:

```python
import os
os.environ["SCIPY_ARRAY_API"] = "1"  # Enable SciPy's array API features
from jax import Array
from scipy.spatial.transform import Rotation as R

def integrate_ang_vel(rot: R, omega: Array, dt: float) -> R:
    return rot * R.from_rotvec(omega * dt)
```

Here, the exponential mapping is implemented via `Rotation.from_rotvec` and composition uses `Rotation` multiplication. Because the entire update is expressed with array API operations, autodiff tracks gradients through integration steps in JAX and PyTorch without custom adjoints.

We use this differentiable simulator to optimize trajectories for imitation learning in drone flight. Example trajectories can be seen in Fig. 3. Gradient-based trajectory optimization adjusts control sequences to minimize a task loss to achieve time optimal flight [11], backpropagating through the rotational dynamics integrated with `Rotation`. This yields optimized trajectories that can, for example, be used as task-aligned supervision signals for policy learning.

4 Conclusion

We overhauled SciPy's `spatial.transform` to be array API compliant and framework-agnostic while preserving its API, delivering rigorously tested $\mathrm{SO}(3)$/$\mathrm{SE}(3)$ primitives that run with NumPy, JAX, PyTorch, CuPy, and others. The new design enables GPU/TPU execution, JIT compatibility, native autodiff, and efficient broadcasting, removing the need for individual, framework-specific reimplementations. Case studies on performance scaling and differentiable drone dynamics demonstrate portability and correctness across devices and frameworks. We invite researchers to incorporate this new capability in their research on differentiable systems whenever they require gradients of rigid transforms or rotations. The implementation is merged and will ship in the next SciPy release.

Acknowledgments and Disclosure of Funding

We thank Lucas Colley, Guido Imperiale and Scott Shambaugh for their input and thorough review of the code. This work was supported by the Robotics Institute Germany under BMFTR grant 16ME0997K, and by the Humboldt Professorship for Robotics and Artificial Intelligence.

References

[1] T. D. Barfoot. *State Estimation for Robotics*. Cambridge University Press, 2017.

[2] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[3] G. Chirikjian, A. Kyatkin, and A. Buckingham. Engineering applications of noncommutative harmonic analysis: With emphasis on rotation and motion groups. *Applied Mechanics Reviews*, 54(6):B97–B98, 11 2001.

[4] J. B. Kuipers. *Quaternions and rotation sequences: A primer with applications to orbits, aerospace, and virtual reality*. Princeton Univ. Press, Princeton, NJ, 1999.

[5] R. M. Murray, S. S. Sastry, and L. Zexiang. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, Inc., USA, 1st edition, 1994.

[6] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[8] Python Data API Consortium. Python array api standard, 2023. Version 2023.12.

[9] N. Ravi, J. Reizenstein, D. Novotny, T. Gordon, W.-Y. Lo, J. Johnson, and G. Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501*, 2020.

[10] E. Riba, D. Mishkin, D. Ponsa, E. Rublee, and G. Bradski. Kornia: an open source differentiable computer vision library for pytorch. In *Winter Conference on Applications of Computer Vision*, 2020.

[11] A. Romero, S. Sun, P. Foehn, and D. Scaramuzza. Model predictive contouring control for time-optimal quadrotor flight. *IEEE Transactions on Robotics*, 38(6):3340–3356, 2022.

[12] E. Todorov, T. Erez, and Y. Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[13] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.