The Journal of Open Source Software

FastTanhSinhQuadrature.jl: High-performance Tanh-Sinh numerical integration in Julia

Stamatis Vretinaris [1,2,3]

1 Institute for Mathematics, Astrophysics and Particle Physics, Radboud University, Heyendaalseweg 135, 6525 AJ Nijmegen, The Netherlands **2** Albert-Einstein-Institut, Max-Planck-Institut für Gravitationsphysik, Callinstraße 38, 30167 Hannover, Germany **3** Leibniz Universität Hannover, 30167 Hannover, Germany

DOI: 10.21105/joss.10076

Software

- Review
- Repository
- Archive

Editor: Vissarion Fisikopoulos

Reviewers:

- @mongibellili
- @ranocha

Submitted: 29 January 2026
Published: 16 April 2026

Summary

Numerical integration is a cornerstone of scientific computing, essential for evaluating integrals that cannot be solved analytically. The Tanh-Sinh (or Double Exponential) quadrature, originally proposed by Takahasi & Mori (1973), is a powerful technique known for its high accuracy and efficiency, particularly for integrands with endpoint singularities. `FastTanhSinhQuadrature.jl` provides a high-performance, arbitrary-precision implementation of this method in Julia. It leverages modern compiler technologies to achieve significant speedups over traditional implementations while maintaining rigorous mathematical precision.

Statement of Need

In many fields of physics and engineering, researchers encounter integrals with singularities at the boundaries. Standard Gaussian quadrature rules often fail or require an excessive number of points to converge in these cases. While other integration libraries exist, they typically lack a combination of robustness, performance, and flexibility. `FastTanhSinhQuadrature.jl` addresses these needs by:

1. **Robustness**: Automatically handling endpoint singularities without manual coordinate transformations.
2. **Performance**: Utilizing SIMD (Single Instruction, Multiple Data) instructions for rapid evaluation.
3. **Flexibility**: Supporting arbitrary precision types (e.g., `BigFloat`) and multidimensional integration.

This package implements a rigorous Tanh-Sinh scheme with an optimized "window selection" strategy enabling SIMD-accelerated execution paths, making it ideal for large-scale simulations where both speed and precision are critical.

State of the field

Numerical integration is a well-established field, with mature implementations in libraries such as **Boost** (Boost C++ Libraries, 2024) (C++), **SciPy** (Virtanen et al., 2020) (Python), and **mpmath** (Johansson & others, 2023) (Python). Within the Julia ecosystem, packages like `QuadGK.jl` (Johnson, 2013) and `HCubature.jl` (Johnson, 2017) provide detailed adaptive Gauss-Kronrod and h-adaptive cubature methods. However, high-performance implementations specifically of the **Tanh-Sinh quadrature** are less common.

Most existing Tanh-Sinh implementations rely on dynamic convergence checks within the summation loop. This introduces conditional branching that prevents modern compilers from applying SIMD vectorization, restricting solvers to scalar execution speeds. `FastTanhSinhQuadrature.jl` overcomes this by adopting the "window selection" strategy described by Vanherck et al. (2020). By analytically pre-calculating the optimal step size h and truncation index n based on floating-point precision, the algorithm eliminates runtime checks, creating a branch-free inner loop amenable to optimization by `LoopVectorization.jl` (Elrod, n.d.).

Mathematics

The Tanh-Sinh quadrature computes integrals of the form $I = \int_{-1}^{1} f(x)\,\mathrm{d}x$ by applying the variable transformation $x = \tanh(\frac{\pi}{2}\sinh(t))$ proposed by Takahasi & Mori (1973). This maps the finite interval to the real line, where the integrand decays double-exponentially. The integral is then approximated using the trapezoidal rule over the infinite domain, truncated to a finite window $[-n, n]$.

For a detailed derivation of the quadrature weights, error bounds, and the window selection strategy used to determine h and n, the reader is referred to Takahasi & Mori (1973) and Vanherck et al. (2020).

Software Design

The package balances ease of use with maximum performance through a two-tier API:

1. **High-Level API** (`quad`): A drop-in replacement for standard quadrature functions, with adaptive stopping based on mixed tolerances (`rtol`, `atol`) and optional reusable caches.
2. **Low-Level API** (`integrateND_avx`): Allows users to pre-compute quadrature nodes and weights for reuse across millions of integrals, eliminating allocation overhead in tight loops.

Key implementation features include:

- **Window Selection**: Uses the method of Vanherck et al. (2020) to pre-determine integration bounds, enabling branch-free loops.
- **SIMD Optimization**: Leverages `LoopVectorization.jl` to vectorize evaluation loops, yielding 2-3x speedups over scalar codes.
- **Reusable Adaptive Caches**: Exposes `adaptive_cache_1D/2D/3D` to avoid repeated cache construction in repeated adaptive workloads.
- **Callable Integrands**: Supports generic callable objects (e.g., struct functors), not just `Function`.
- **Static Allocation**: For moderate node counts, weights and nodes can be stored in `StaticArrays`, eliminating heap allocations.
- **Arbitrary Precision**: Supports generic number types (`BigFloat`, `Double64`, `Float64x2`) by dynamically deriving quadrature parameters from machine epsilon.

Research Impact

`FastTanhSinhQuadrature.jl` has been integrated as a backend for `Integrals.jl` (Rackauckas & Nie, 2017), ensuring widespread availability within the SciML ecosystem.

To support long-term reproducibility, the software snapshot used for this manuscript is archived on Zenodo (Vretinaris, 2026). Command-level reproduction steps (tests, benchmarks, figure generation, and draft PDF workflow) are documented in the repository's `REPRODUCIBILITY.md` guide.

Performance

Figure 1 summarizes benchmarks against `FastGaussQuadrature.jl` (Townsend et al., 2013), `QuadGK.jl` (Johnson, 2013), `HCubature.jl` (Johnson, 2017), `Cubature.jl` (Johnson, 2005), and `Cuba.jl` (Hahn, 2005, 2015). All benchmarks use `rtol = 10^{-6}` and `atol = 10^{-8}`; external adaptive solvers are capped at 200,000 evaluations. For each benchmark case, the plotted speedup is measured relative to the fastest competing method that also met the requested tolerance.

The results show two distinct usage regimes. The high-level quad interface is most compelling for endpoint-singular integrands: in the reproduced benchmark run it is about **18x** faster than `QuadGK.jl` on $(1-x^2)^{-1/2}$ and about **6x** faster on $\log(1-x)$, while in the tested 2D endpoint-singular case it is more than three orders of magnitude faster than the fastest accurate alternative. The SIMD path (`integrate*_avx`) is the main performance-oriented API: in the directly comparable benchmark set it is the fastest accurate method in a majority of cases, and it remains competitive or superior across many singular and smooth tensor-product problems.

Because SIMD behavior can depend on Julia and `LoopVectorization` compatibility, Figure 1 reports two SIMD markers: one for the newer-Julia path and one for Julia 1.12 (where the fully active `LoopVectorization` path is available).

These benchmarks also clarify the package's limitations. `FastTanhSinhQuadrature.jl` should be preferred when the integrand has endpoint singularities, when the same quadrature rule can be reused across many evaluations, or when arbitrary precision is required. It is less advantageous for smooth low-dimensional problems where specialized Gauss or Gauss-Kronrod rules already match the integrand well. For example, `FastGaussQuadrature.jl` and `QuadGK.jl` are faster on the smooth 1D polynomial and Runge-function tests, and `Cuba.jl`/`Cubature.jl` can outperform the adaptive quad interface on some smooth 3D problems. Interior singularities are likewise not handled automatically and still require domain splitting via `quad_split`.

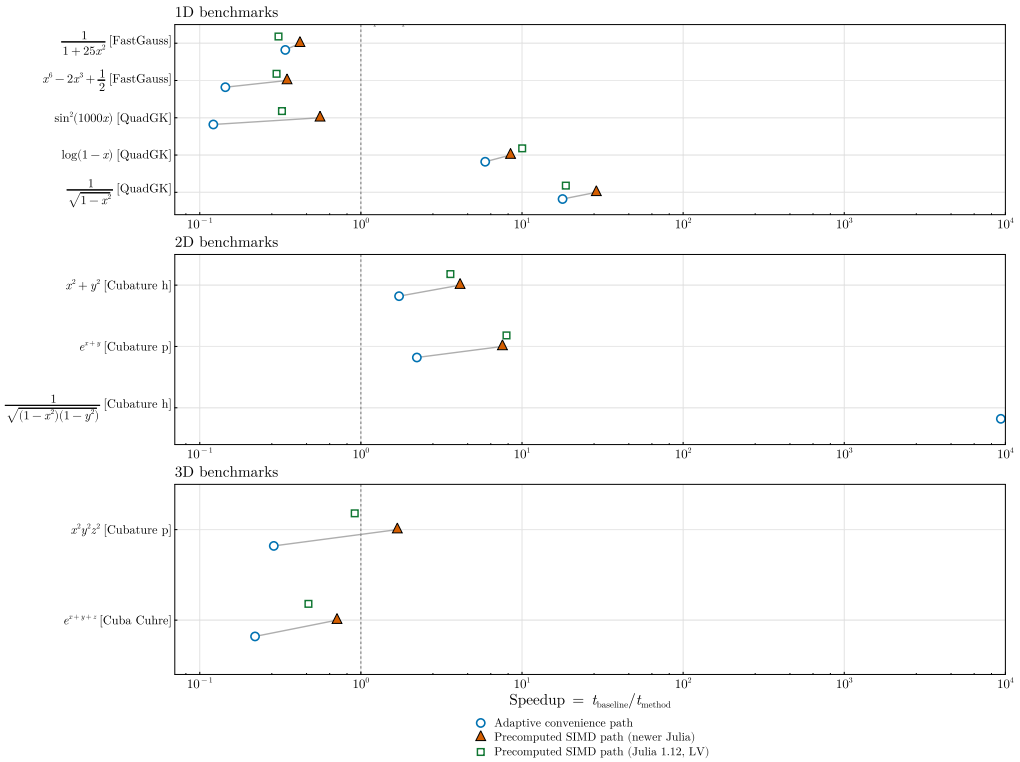

Figure 1: Figure 1. Speedup of `quad` and `integrate*_avx` relative to the fastest accurate competing method on each benchmark problem. Two SIMD markers are shown: newer-Julia and Julia 1.12 (`LoopVectorization`-active). The 3D endpoint-singular case is omitted because `FastTanhSinhQuadrature.jl` was the only tested method that satisfied the requested tolerance.

Usage

Basic Integration

```julia
using FastTanhSinhQuadrature

# Integrate exp(x) from 0 to 1
val = quad(exp, 0.0, 1.0; rtol=1e-8, atol=1e-10) # ≈ 1.71828...

# Handle singularities: 1/sqrt(x)
val = quad(x -> 1/sqrt(x), 0.0, 1.0) # ≈ 2.0

# Reuse an adaptive cache across repeated calls
cache = adaptive_cache_1D(Float64; max_levels=16)
val = quad(exp, 0.0, 1.0; rtol=1e-8, atol=1e-10, cache=cache)
```

High-Performance Pre-computation

```julia
# Pre-compute nodes/weights for Float64
x, w, h = tanhsinh(Float64, Val(80))

# Reuse in tight loops (zero-allocation)
f(t) = sin(t)^2
integral = integrate1D_avx(f, 0.0, π, x, w, h)
```

Convergence

Convergence tests for various integrands are shown below. The method exhibits rapid exponential convergence characteristic of the Tanh-Sinh scheme.

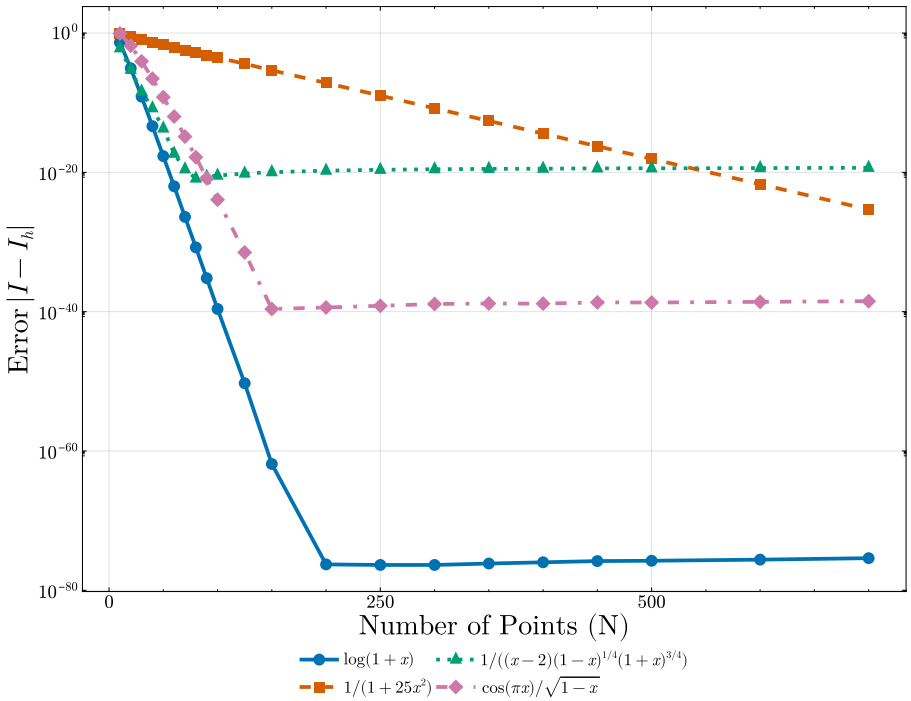

Figure 2: Convergence of Tanh-Sinh Quadrature compared to other methods.

Future Work

Natural directions for future development include extending the current window-selection strategy to additional exponential and double-exponential quadrature formulas, deriving the corresponding window sizes so that SIMD-friendly execution can be enabled for a broader family of rules, and adding parallel execution paths such as multithreading. Another clear next step is to generalize the current specialized 1D-3D routines to an n-dimensional formulation for arbitrary tensor-product dimensions.

AI usage disclosure

During the development of this package, the author utilized Gemini (Google) for assistance with documentation, debugging and the generation of the first draft of this paper. The author has reviewed and edited all AI-generated content to ensure accuracy and adherence to the package's coding standards.

Acknowledgements

The author acknowledges the developers of `LoopVectorization.jl` for providing the tools that enabled the performance optimizations in this package.

References

Boost C++ Libraries. (2024). *Boost C++ Libraries*. https://www.boost.org/

Elrod, C. (n.d.). *LoopVectorization.jl*. https://github.com/JuliaSIMD/LoopVectorization.jl

Hahn, T. (2005). Cuba: A library for multidimensional numerical integration. *Computer Physics Communications*, *168*, 78–95. https://doi.org/10.1016/j.cpc.2005.01.010

Hahn, T. (2015). Concurrent Cuba. *Journal of Physics: Conference Series*, *608*(1), 012066. https://doi.org/10.1088/1742-6596/608/1/012066

Johansson, F., & others. (2023). *mpmath: A Python library for arbitrary-precision floating-point arithmetic*. http://mpmath.org/

Johnson, S. G. (2005). *Multi-dimensional adaptive integration in C: The Cubature package*. https://github.com/stevengj/cubature.

Johnson, S. G. (2013). *QuadGK.jl: Gauss–Kronrod integration in Julia*. https://github.com/JuliaMath/QuadGK.jl.

Johnson, S. G. (2017). *The HCubature.jl package for multi-dimensional adaptive integration in Julia*. https://github.com/JuliaMath/HCubature.jl.

Rackauckas, C., & Nie, Q. (2017). DifferentialEquations.jl – a performant and feature-rich ecosystem for solving differential equations in Julia. *The Journal of Open Research Software*, *5*(1). https://doi.org/10.5334/jors.151

Takahasi, H., & Mori, M. (1973). Double exponential formulas for numerical integration. *Publications of the Research Institute for Mathematical Sciences*, *9*(3), 721–741. https://doi.org/10.2977/PRIMS/1195192451

Townsend, A., Hale, N., & Olver, S. (2013). *FastGaussQuadrature.jl*. https://github.com/JuliaApproximation/FastGaussQuadrature.jl

Vanherck, J., Sorée, B., & Magnus, W. (2020). Tanh-sinh quadrature for single and multiple integration using floating-point arithmetic. *arXiv Preprint arXiv:2007.15057*. https://doi.org/10.48550/arXiv.2007.15057

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Vretinaris, S. (2026). *FastTanhSinhQuadrature.jl*. https://zenodo.org/records/19420466. https://doi.org/10.5281/zenodo.19420466