




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,
² 6525 AJ Nijmegen, The Netherlands ² Albert-Einstein-Institut, Max-Planck-Institut für
³ Gravitationsphysik, Callinstraße 38, 30167 Hannover, Germany ³ Leibniz Universität Hannover, 30167
Hannover, Germany

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Open Journals](#) 

Reviewers:

- [@openjournals](#)

Submitted: 01 January 1970

Published: unpublished

License

Authors of papers retain copyright
and release the work under a
Creative Commons Attribution 4.0
International License ([CC BY 4.0](#))

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.

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

46 Mathematics

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

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

55 Software Design

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

- 57 1. **High-Level API** (quad): A drop-in replacement for standard quadrature functions,
58 handling adaptivity, singularities, and infinite domains automatically.
- 59 2. **Low-Level API** (integrateND_avx): Allows users to pre-compute quadrature nodes and
60 weights for reuse across millions of integrals, eliminating allocation overhead in tight
61 loops.

62 Key implementation features include:

- 63 ■ **Window Selection:** Uses the method of Vanherck et al. (2020) to pre-determine
64 integration bounds, enabling branch-free loops.
- 65 ■ **SIMD Optimization:** Leverages LoopVectorization.jl to vectorize evaluation loops,
66 yielding 2-3x speedups over scalar codes.
- 67 ■ **Static Allocation:** For moderate node counts, weights and nodes can be stored in
68 StaticArrays, eliminating heap allocations.
- 69 ■ **Arbitrary Precision:** Supports generic number types (BigFloat, Double64, Float64x2)
70 by dynamically deriving quadrature parameters from machine epsilon.

71 Research Impact

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

74 Performance

75 Figure 1 summarizes benchmarks against FastGaussQuadrature.jl (Townsend et al., 2013),
76 QuadGK.jl (Johnson, 2013), HCubature.jl (Johnson, 2017), Cubature.jl (Johnson, 2005),
77 and Cuba.jl (Hahn, 2005, 2015). All benchmarks use $\text{rtol} = 10^{-6}$ and $\text{atol} = 10^{-8}$;
78 external adaptive solvers are capped at 200,000 evaluations. For each benchmark case, the

79 plotted speedup is measured relative to the fastest competing method that also met the
80 requested tolerance.

81 The results show two distinct usage regimes. The high-level quad interface is most compelling
82 for endpoint-singular integrands: in 1D it is about **6.7x** faster than QuadGK.jl on $(1 - x^2)^{-1/2}$
83 and about **2.2x** faster on $\log(1 - x)$, while in the tested 2D endpoint-singular case it is more
84 than three orders of magnitude faster than the fastest accurate alternative. The SIMD path
85 (`integrate*_avx`) is the main performance-oriented API: it is the fastest accurate method in
86 7 of the 9 directly comparable benchmarks, and it remains competitive or superior across many
87 singular and smooth tensor-product problems.

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

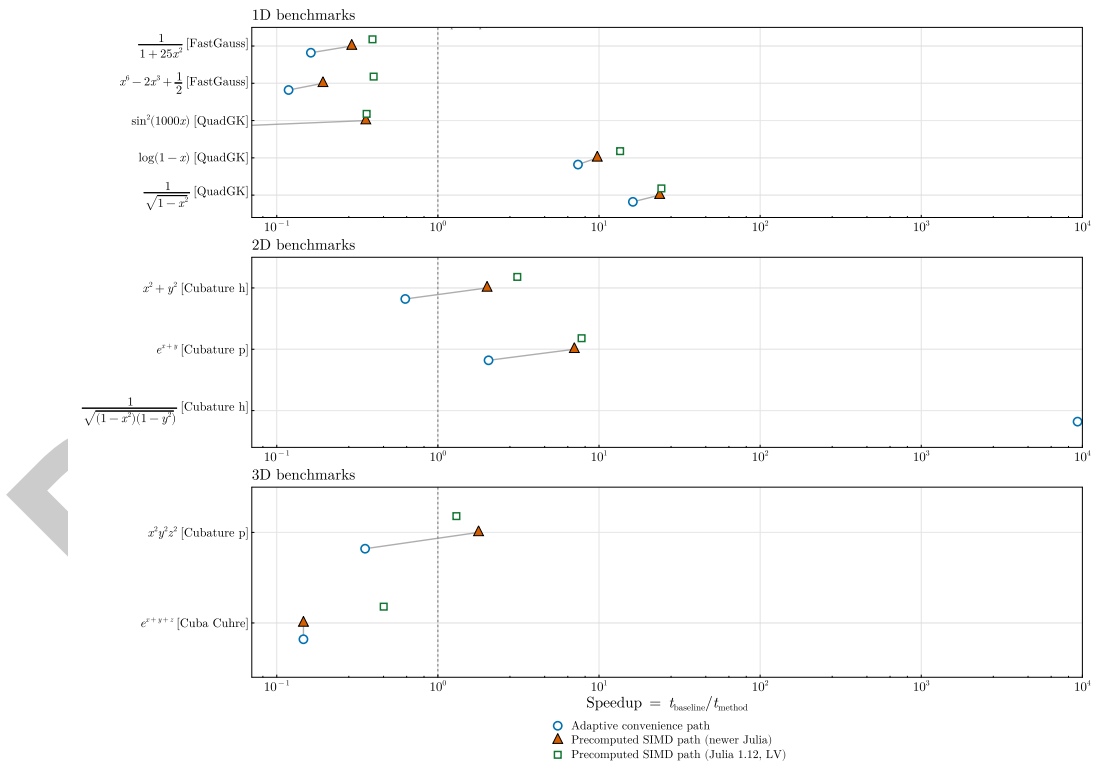


Figure 1: Figure 1. Speedup of quad and `integrate*_avx` relative to the fastest accurate competing method on each benchmark problem. The 3D endpoint-singular case is omitted because `FastTanhSinhQuadrature.jl` was the only tested method that satisfied the requested tolerance.

96 **Usage**

97 **Basic Integration**

using FastTanhSinhQuadrature

```
# Integrate exp(x) from 0 to 1
val = quad(exp, 0.0, 1.0) # ≈ 1.71828...

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

98 High-Performance Pre-computation

```
# 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)
```

99 Convergence

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

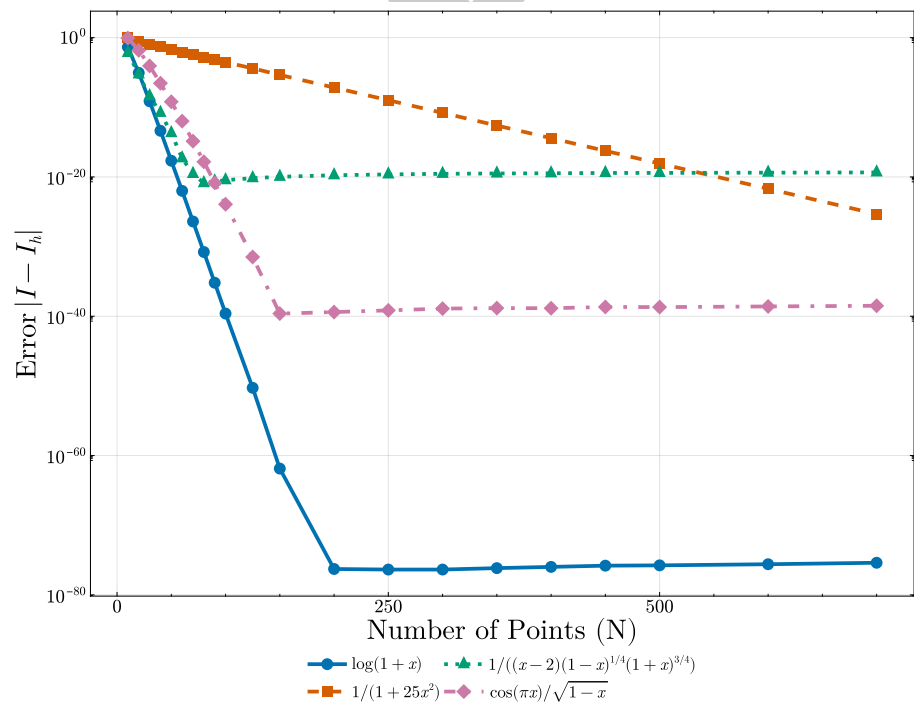


Figure 2: Convergence of Tanh-Sinh Quadrature compared to other methods.

102 Future Work

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

109 AI usage disclosure

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

114 Acknowledgements

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

117 References

- 118 Boost C++ Libraries. (2024). *Boost C++ Libraries*. <https://www.boost.org/>
- 119 Elrod, C. (n.d.). *LoopVectorization.jl*. <https://github.com/JuliaSIMD/LoopVectorization.jl>
- 120 Hahn, T. (2005). Cuba: A library for multidimensional numerical integration. *Computer*
121 *Physics Communications*, 168, 78–95. <https://doi.org/10.1016/j.cpc.2005.01.010>
- 122 Hahn, T. (2015). Concurrent Cuba. *Journal of Physics: Conference Series*, 608(1), 012066.
123 <https://doi.org/10.1088/1742-6596/608/1/012066>
- 124 Johansson, F., & others. (2023). *mpmath: A Python library for arbitrary-precision floating-*
125 *point arithmetic*. <http://mpmath.org/>
- 126 Johnson, S. G. (2005). *Multi-dimensional adaptive integration in C: The Cubature package*.
127 <https://github.com/stevengj/cubature>.
- 128 Johnson, S. G. (2013). *QuadGK.jl: Gauss–Kronrod integration in Julia*. <https://github.com/JuliaMath/QuadGK.jl>.
- 129 Johnson, S. G. (2017). *The HCubature.jl package for multi-dimensional adaptive integration*
130 *in Julia*. <https://github.com/JuliaMath/HCubature.jl>.
- 131 Rackauckas, C., & Nie, Q. (2017). DifferentialEquations.jl – a performant and feature-rich
132 ecosystem for solving differential equations in Julia. *The Journal of Open Research Software*,
133 5(1). <https://doi.org/10.5334/jors.151>
- 134 Takahasi, H., & Mori, M. (1973). Double exponential formulas for numerical integration.
135 *Publications of the Research Institute for Mathematical Sciences*, 9(3), 721–741. <https://doi.org/10.2977/PRIMS/1195192451>
- 136 Townsend, A., Hale, N., & Olver, S. (2013). *FastGaussQuadrature.jl*. <https://github.com/JuliaApproximation/FastGaussQuadrature.jl>
- 137 Vanherck, J., Sorée, B., & Magnus, W. (2020). Tanh-sinh quadrature for single and multiple
138 integration using floating-point arithmetic. *arXiv Preprint arXiv:2007.15057*. <https://doi.org/10.48550/arXiv.2007.15057>
- 139 Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D.,
140 Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson,
141 J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., ... SciPy
142 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in
143 Python. *Nature Methods*, 17, 261–272. <https://doi.org/10.1038/s41592-019-0686-2>