The Journal of Open Source Software

modepy: Basis Functions, Interpolation, and Quadrature (not just) for Finite Elements

Andreas Kloeckner [1], **Alexandru Fikl** [2], **Xiaoyu Wei** [3], **Thomas Gibson** [4], **and Addison Alvey-Blanco**[1]

1 Siebel School of Computing and Data Science, University of Illinois at Urbana-Champaign, US **2** Institute for Advanced Environmental Research (ICAM), West University of Timișoara, Romania **3** Pathlit, US **4** Advanced Micro Devices Inc., US

DOI: 10.21105/joss.09294

Software

- Review
- Repository
- Archive

Editor: Nikoleta Glynatsi

Reviewers:

- @capitalaslash
- @FranziLesi

Submitted: 03 September 2025
Published: 27 January 2026

Summary

`modepy` is a Python library for defining reference elements, equipping them with appropriate approximation spaces, and numerically performing calculus operations (derivatives, integrals) on those spaces. It is written in pure, type-annotated Python 3, offering comprehensive documentation and minimal runtime dependencies (mainly NumPy).

`modepy` focuses on high-order accuracy — given an element size h, this refers to the asymptotic decay of the approximation error as $O(h^n)$, for $n \geq 3$, assuming sufficient smoothness of the solution being approximated. For a problem in d dimensions, the number of unknowns scales as $O(h^{-d})$. Therefore, if accuracy is desired at manageable cost, high-order methods are crucial.

A popular approach for accurate approximation of functions on geometrically complex domains is the use of *unstructured discretizations*, e.g. in the Finite Element Method (FEM). The geometry is typically represented as a disjoint union (a "mesh") of primitive geometric shapes, most often simplices and quadrilaterals. Given the means to perform calculus operations on these *reference elements* and mapping functions from them to the *global* elements, calculus operations become available on the entire domain. These primitives are chiefly useful in the numerical solution of integral and (partial) differential equations. Additional applications include computer graphics, Computer Aided Design (CAD), and robotics. Those, in turn, can be used to model many physical phenomena, including fluid flow, electromagnetism, and solid mechanics. `modepy` has been used to construct FEM solvers (Glusa, 2021; Klöckner & others, 2025a) and integral equation solvers (Klöckner & others, 2025b) that run on both CPUs and GPUs.

Statement of need

The functionality outlined above is often embedded in an ad-hoc manner in larger codes, restricting scope and reusability. `modepy` addresses this need by providing a reusable, generalizable, and composable implementation.

There are several other libraries in the literature with similar goals, but important differences and limitations. FInAT (Ham et al., 2025) (and the earlier FIAT (Homolya et al., 2025)) offers reference elements and basis functions, but is tightly coupled to the FEniCS/Firedrake ecosystem. Similarly, `StartUpDG.jl` (Chan et al., 2024) has a focus on the needs of discontinuous Galerkin FEM in the Trixi framework. `QuadPy` (Schlömer et al., 2021) provides access to quadrature rules, but it is no longer open source and lacks `modepy`'s composability. `minterpy` (Wicaksono et al., 2025), meanwhile, deals exclusively with polynomial interpolation, with a focus on sparse grids.

The solvers served by `modepy` typically have tight cost constraints, often adopting HPC techniques (GPU, MPI, etc.). To facilitate separation of implementation and high-performance concerns from the core numerical method, `modepy` adopts a two-pronged approach. First, if it suffices to represent operations as data in matrix or tabular form, execution of `modepy` code is not needed in a cost-constrained setting. For example, nodes and bilinear forms on reference elements can generally be pre-computed and tabulated. Second, if this tabulation approach falls short, `modepy` provides data structures to reveal additional internal structure.

Tensor product elements provide an example of this. In this instance, many operator matrices permit a Kronecker product factorization that significantly reduces the asymptotic complexity of a matrix-vector product in higher dimensions (Orszag, 1980). `modepy` exposes functionality that allows reshaping degrees of freedom arrays to take advantage of such factorizations. Another prominent example is the evaluation of basis functions at points known only at runtime. To facilitate efficient evaluation, `modepy` allows its functions to be "traced", in the sense of lazy or deferred evaluation. The resulting expression graph is represented by the `pymbolic` (Klöckner et al., 2024) software library, that can interoperate with Python ASTs (Python Software Foundation, 2024), SymPy (Meurer et al., 2017), SymEngine (Čertík et al., 2013), etc., for straightforward generation of high-performance code.

Overview

The high-level concepts available in `modepy` are shapes (i.e. reference domains), modes (i.e. the basis functions), and nodes (i.e. the degrees of freedom). These are implemented in a user-extensible fashion using the `singledispatch` mechanism, with inspiration taken from common idiomatic usage in Julia (Bezanson et al., 2017).

Shapes

The geometry of a reference element is described in `modepy` by the `Shape` class. Built-in support exists for `Simplex` and `Hypercube` geometries, encompassing the commonly used interval, triangle, tetrahedron, quadrilateral, and hexahedral shapes (see Figure 1). `TensorProductShape` can be used to compose additional shapes (e.g. prims, as generated by, e.g. `gmsh` (Geuzaine & Remacle, 2009)).

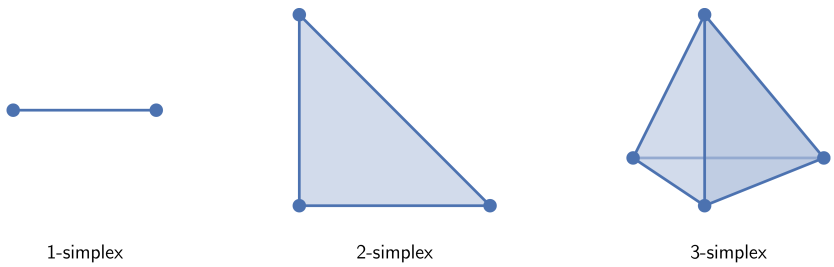

Figure 1: Domains corresponding to the one-, two-, and three-dimensional simplices.

Modes and Spaces

To perform calculus operations, each reference element can be equipped with a function space described by the `FunctionSpace` class. These represent a finite-dimensional space of functions $\phi_i : D \to \mathbb{R}$, where D is the reference element domain, and no specific choice of basis. Predefined choices include the `PN` space, containing polynomials of total degree at most N, and the `QN` space, containing polynomials of maximum degree at most N. As with shapes, these spaces can be combined using `TensorProductSpace`. A `Basis` objects is available separately, giving access to basis functions and their derivatives, for, e.g., the monomials, general Jacobi polynomials, and the Proriol-Koornwinder-Dubiner-Owens (PKDO) basis from (Dubiner, 1991) (see Figure 2).

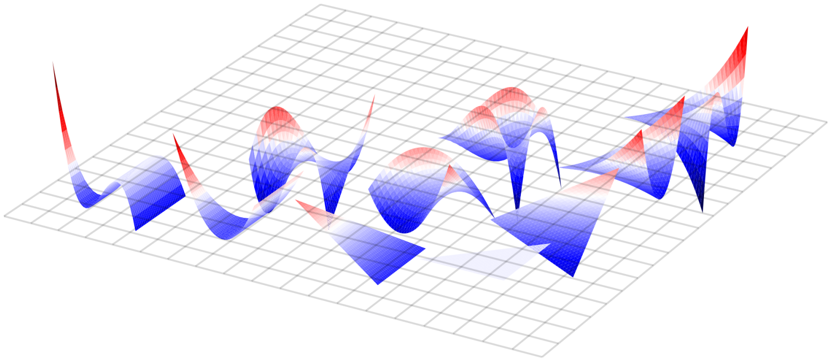

Figure 2: PKDO basis functions for the triangle.

Nodes

A final component in an FEM discretization (Brenner & Scott, 2007, sec. 3.1) is a set of 'degrees of freedom' ('DOFs') that uniquely identify a certain function in the span of a basis. `modepy` supports modal DOFs (i.e. basis coefficients) and nodal DOFs (i.e. function or derivative values at a point). On simplices, the "warp-and-blend" nodes (Warburton, 2007) are available, and on the hypercube, standard tensor product nodes are constructed from one-dimensional Legendre-Gauss(-Lobatto) nodes. `modepy` can also directly interoperate with the `recursivenodes` library described in (Isaac, 2020), which offers additional well-conditioned nodes on the simplex.

Quadrature

`modepy` also offers a wide array of quadrature rules that can be used on each reference element. For the interval, Clenshaw–Curtis, Fejér, and Jacobi-Gauss(-Lobatto) are provided. Many more state-of-the-art rules are available, typically up to high order $n > 20$ from (Grundmann & Möller, 1978; Jaśkowiec & Sukumar, 2021; Vioreanu & Rokhlin, 2014; Witherden & Vincent, 2015; Xiao & Gimbutas, 2010) (see Figure 3). There is also functionality (Vioreanu & Rokhlin, 2014) to allow constructing novel quadratures on a given domain.

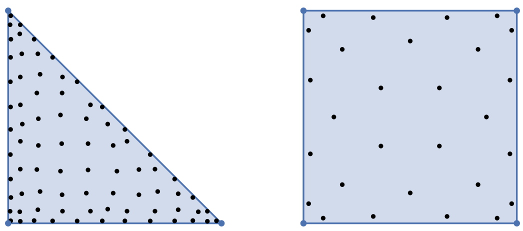

Figure 3: (left) Vioreanu–Rokhlin quadrature points of order 11 and (right) Witherden–Vincent quadrature points of order 11.

Matrices

`modepy`'s functionality is rounded out by various tabulation and matrix generation functions. This includes the ability to tabulate operator matrices for fairly general bilinear forms used in FEM.

Acknowledgements

A. Fikl was supported by the Office of Naval Research (ONR) as part of the Multidisciplinary University Research Initiatives (MURI) Program, under Grant Number *N00014-16-1-2617*. A. Klöckner was supported by the US National Science Foundation under award number DMS-2410943, and by the US Department of Energy under award number DE-NA0003963.

References

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*, 65–98. https://doi.org/10.1137/141000671

Brenner, S., & Scott, R. (2007). *The mathematical theory of finite element methods*. Springer Science & Business Media. ISBN: 9780387759333

Čertík, O., Fernando, I., Garg, S., Rathnayake, T., & others. (2013). *SymEngine: A fast symbolic manipulation library written in C++*. https://github.com/symengine/symengine

Chan, J., Knapp, D., McCallum, M., Ranocha, H., Wang, V. X., & Markert, J. (2024). *StartUpDG.jl: Reference elements and physical meshes for DG* (Version v1.1.5). https://github.com/jlchan/StartUpDG.jl

Dubiner, M. (1991). Spectral methods on triangles and other domains. *Journal of Scientific Computing*, *6*, 345–390. https://doi.org/10.1007/bf01060030

Geuzaine, C., & Remacle, J.-F. (2009). Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International Journal for Numerical Methods in Engineering*, *79*(11), 1309–1331. https://doi.org/10.1002/nme.2579

Glusa, C. (2021). *PyNucleus: A finite element code that specifically targets nonlocal operators*. Sandia National Lab (SNL-NM), Albuquerque, NM (United States). https://github.com/sandialabs/PyNucleus

Grundmann, A., & Möller, H. M. (1978). Invariant integration formulas for the n-simplex by combinatorial methods. *SIAM Journal on Numerical Analysis*, *15*, 282–290. https://doi.org/10.1137/0715019

Ham, D. A., Homolya, M., Kirby, R., Mitchell, L., Brubeck, P., cyruscycheng21, FAznaran, ksagiyam, Scroggs, M., Farrell, P. E., Justincrum, Ward, C., celdred, Bendall, T., Betteridge, J., & FabianL1908. (2025). *FInAT: A smarter library of finite elements* (Version Firedrake_20250331.0). Zenodo. https://doi.org/10.5281/zenodo.15114385

Homolya, M., Brubeck, P., Ham, D. A., Kirby, R., Mitchell, L., Blechta, J., Rognes, M. E., Wells, G. N., Logg, A., cyruscycheng21, Gibson, T. H., Ring, J., Yashchuk, I., Justincrum, k-b-oelgaard, ksagiyam, Richardson, C., mer, N. S., Nixon-Hill, R. W., … Scroggs, M. (2025). *FIAT: 2025.4.0* (Version 2025.4.0). Zenodo. https://doi.org/10.5281/zenodo.15302339

Isaac, T. (2020). Recursive, parameter-free, explicitly defined interpolation nodes for simplices. *SIAM Journal on Scientific Computing*, *42*, A4046–a4062. https://doi.org/10.1137/20m1321802

Jaśkowiec, J., & Sukumar, N. (2021). High-order symmetric cubature rules for tetrahedra and pyramids. *International Journal for Numerical Methods in Engineering*, *122*, 148–171. https://doi.org/10.1002/nme.6528

Klöckner, A., & others. (2025a). *Grudge: An environment for discontinuous Galerkin discretizations* (Version hash:193100a). https://github.com/inducer/grudge

Klöckner, A., & others. (2025b). *Pytential: Evaluate layer and volume potentials accurately* (Version hash:7395b97f). https://github.com/inducer/pytential

Klöckner, A., Wala, M., Fernando, I., Kulkarni, K., Fikl, A., Weiner, Z., Kempf, D., Ham, D. A., Mitchell, L., Wilcox, L. C., Diener, M., Kapyshin, P., Raksi, R., & Gibson, T. H. (2024). *Pymbolic: A package to do symbolic operations for code generation* (Version v2024.2.2). Zenodo. https://doi.org/10.5281/zenodo.14526145

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., & others. (2017). SymPy: Symbolic computing in Python. *PeerJ Computer Science*, *3*, e103.

Orszag, S. A. (1980). Spectral methods for problems in complex geometries. *Journal of Computational Physics*, *37*, 70–92. https://doi.org/10.1016/0021-9991(80)90005-4

Python Software Foundation. (2024). *ast — abstract syntax trees*. https://docs.python.org/3/library/ast.html

Schlömer, N., Papior, N., Arnold, D., Blechta, J., & Zetter, R. (2021). *QuadPy: Numerical integration (quadrature, cubature) in Python* (Version v0.16.10). Zenodo. https://doi.org/10.5281/zenodo.5541216

Vioreanu, B., & Rokhlin, V. (2014). Spectra of multiplication operators as a numerical tool. *SIAM Journal on Scientific Computing*, *36*, A267–a288. https://doi.org/10.1137/110860082

Warburton, T. (2007). An explicit construction of interpolation nodes on the simplex. *Journal of Engineering Mathematics*, *56*, 247–262. https://doi.org/10.1007/s10665-006-9086-6

Wicaksono, D., Acosta, U. H., Veettil, S. K. T., Kissinger, J., & Hecht, M. (2025). Minterpy: Multivariate polynomial interpolation in Python. *Journal of Open Source Software*, *10*(109), 7702. https://doi.org/10.21105/joss.07702

Witherden, F. D., & Vincent, P. E. (2015). On the identification of symmetric quadrature rules for finite element methods. *Computers & Mathematics With Applications*, *69*, 1232–1241. https://doi.org/10.1016/j.camwa.2015.03.017

Xiao, H., & Gimbutas, Z. (2010). A numerical algorithm for the construction of efficient quadrature rules in two and higher dimensions. *Computers & Mathematics With Applications*, *59*, 663–676. https://doi.org/10.1016/j.camwa.2009.10.027