The Journal of Open Source Software

pyaudi: A truncated Taylor polynomial algebra toolbox for differentiable intelligence, automatic differentiation, and verified integration applications.

Dario Izzo [1], **Francesco Biscani** [2], **and Sean Cowan** [1]

1 Advanced Concepts Team, European Space Research and Technology Center (Noordwijk, NL) **2** Max Planck Institute for Astronomy (Heidelberg, DE)

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

pyaudi is an open-source Python toolbox developed at the European Space Agency that provides high-order, forward-mode automatic differentiation in a multivariate setting. The toolbox is built on C++ class templates, exposed to Python via pybind11, and, at its core, implements the algebra of truncated Taylor polynomials. This design allows the underlying generalized dual number type to act as a seamless drop-in replacement for scalar types such as floats, while operator overloading ensures that derivatives are propagated automatically. The C++ code base `audi` can also be used directly and allows for greater flexibility in the instantiation of the algebra over different fields.

All standard mathematical operations are supported, leveraging the nilpotency of exponentiation in the algebra of truncated Taylor polynomials. In essence, within a truncated algebra $\mathcal{A}^n$, any polynomial $p \in \mathcal{A}^n$ with vanishing constant term evaluates to zero when raised to a power greater than the truncation order ($m > n$). This property enables efficient and exact computation of derivatives to arbitrary order while maintaining the simplicity of standard numerical code.

On top of the algebra of truncated Taylor polynomials, `pyaudi` also offers an implementation of Taylor models (Makino, 1998), which combines truncated Taylor polynomials with an interval bounding their truncation error as well as a number of miscellaneous algorithms useful for applications in differential intelligence, high-order automatic differentiation, verified integration and more.

Statement of need

`pyaudi` enables users to compute and manipulate order-n Taylor expansions of generic computational graphs, while also providing rigorous bounds on the truncation error through their associated Taylor models. These representations of program outputs can be exploited in a variety of ways, including fast Monte Carlo simulations, rigorous uncertainty analysis, local inversion of output–input relations, and high-order sensitivity studies. The package implements the high-order automatic differentiation methodology originally developed by Berz and Makino ((Berz et al., 2014), (Makino, 1998)), while introducing novel implementation details in polynomial multiplication routines and in the bounding of Taylor models.

Existing libraries with similar capabilities

As of the time of writing, there are two main open source libraries that allow to perform similar computations to those allowed by `pyaudi`. The first one is the C++/C library DACE (Massari et al., 2018) implementing the full differential algebra of truncated Taylor polynomials with float coefficients. Unlike `pyaudi`, DACE relies on a polynomial multiplication routine that makes extensive use of memory for the storage of monomial coefficients. As discussed in the comparison reported below, this approach gives DACE an advantage for single evaluations at lower orders, with the benefit diminishing as computations are performed in batches and at high orderders.

A second relevant project are the Julia libraries TaylorSeries.jl and TaylorModels.jl (Benet et al., 2019) providing implementations of Taylor models to compute rigorous bounds on generic Taylor series. However, their underlying approach differs substantially from that of `pyaudi`, and preliminary comparisons presented here indicate that `pyaudi` can significantly outperform these libraries in the practical cases tested.

Key aspects

The main features of `pyaudi` are:

- **Truncated polynomial algebra**, powered by Obake, a C++ library for symbolic manipulation of sparse multivariate polynomials, truncated power series, and Poisson series. Unlike other packages, which often suffer from severe memory bottlenecks as the polynomial order or the number of variables increases, `pyaudi` avoids large static memory allocations by adopting a sparse, dynamic approach. This remains memory-efficient at the cost of additional bookkeeping, where sparse polynomials are the area of greatest benefit. The use of templates allows to instantite the algebra over different fields such as floats, quadruple precision floats, vectorized floats, etc..
- **Vectorized coefficients**, enabling the simultaneous evaluation of identical computational graphs at multiple expansion points. This feature makes it possible to compute high-order derivatives on multiple points, while amortizing the overhead introduced by the sparse bookkeeping of Obake.
- **Taylor models with Bernstein polynomial bounding**, used to enclose the range of multivariate polynomials.
- **Map inversion algorithm**, implementing the method described in (Berz et al., 2014). This feature enables the local inversion of input–output relations arising in generic computational graphs.

Comparison with DACE

The main difference between the DACE library, with respect to automated differentiation capabilities, and `pyaudi` is to be found in their polynomial multiplication algorithm. We thus focus on that for a preliminary comparison. Our benchmarks run on an AMD EPYC 7702 64-Core Processor with 512GB of RAM. and show the relative computational time of the exact same quantities. We compare the polynomial multiplication algorithm in `pyaudi` with the one implemented in DACE, on a single polynomial multiplication.

We thus intoduce two polynomials of the form:

$$
\begin{aligned}
p_1 &= (1 + x_1 + x_2 + \ldots + x_n)^m \\
p_2 &= (1 - x_1 - x_2 - \ldots - x_n)^m
\end{aligned}
$$

where $x_i, i = 1..n$ etc. are the variables and m the order. The polynomials are then multiplied and the result of $p_1 p_2$ timed. For this simple and basic operation, the speed up of pyaudi w.r.t. DACE is reported in the table below in seconds.

nvars↓ Order→	6	7	8	9	10	11	12	13	14	15
6	0.272	0.119	0.0704	0.145	0.193	0.392	0.389	0.488	0.984	1.13
8	0.152	0.108	0.169	0.399	0.67	1.28	1.41	3.19	6.01	8.76
10	0.205	0.173	0.386	0.637	1.38	2.95	7.15	10.9	19.9	30.2
12	0.134	0.524	0.548	1.47	2	6.09	14.3	24	35	55.7

It can be seen that pyaudi is faster from nvars + order >≈ 19 where memory management becomes an issue. At lower orders and number of variables DACE is significantly faster as its able to expliot an easier memory structure and has no overhead.

Vectorized coefficients

In order to mitigate the bookkeeping overheads of `pyaudi`, vectorized coefficients have been implemented as to allow to perform the same computations over batches, also having in mind potential machine learning applications.

To showcase the resulting performances of such a vectorization, we perform the following computation batching the value c_v of the constant coefficient.

$$p1 = \frac{c_v + x1 + x2 + \ldots + xn}{c_v - x1 - x2 - \ldots - xn}^5$$

It is worth noting here how this operation is not representative of actual applications where one is mostly interested in computing higher order derivatives of computer programs, rather its selected to isolate the feature we are poposing to benchmark which is batching coefficients. In case of DACE we perform the same computation over the entire batch in a loop.

The results are displayed in three tables per number of variables below.

2 variables

points↓ Order →	1	2	3	4	5	6	7	8	9
16	0.256	0.267	0.219	0.208	0.115	0.293	0.23	0.414	0.367
64	1.18	1.36	1.23	0.861	0.954	1.64	1.44	1.67	0.251
256	4.3	4.66	3.79	0.56	0.411	0.677	0.644	0.648	0.68
1024	6.81	0.539	0.776	0.685	0.54	0.745	0.628	0.633	0.632
4096	0.387	0.971	0.734	0.767	0.718	1.03	0.809	0.949	0.932
16384	1.26	1.35	0.998	0.727	0.834	1.07	0.934	1.04	0.9

5 variables

points↓ Order →	1	2	3	4	5	6	7	8	9
16	0.193	0.34	0.273	0.304	0.0819	0.139	0.153	0.164	0.195
64	0.973	1.34	0.192	0.197	0.196	0.348	0.378	0.461	0.552
256	3.77	0.401	0.316	0.31	0.343	0.686	0.793	0.969	1.37
1024	1.22	0.66	0.427	0.425	0.462	0.91	1.01	1.5	1.93
4096	1.52	0.733	0.497	0.545	0.663	0.966	1.22	1.72	2.38
16384	1.86	1.05	0.616	0.616	0.649	1.45	1.76	2.63	2.87

10 variables

points↓ Order →	1	2	3	4	5	6	7	8	9
16	0.513	0.199	0.0438	0.0688	0.0797	0.229	0.291	0.464	0.526
64	1.31	0.138	0.152	0.182	0.262	0.712	0.807	1.51	2.25
256	4.33	0.487	0.388	0.403	0.659	1.8	2.5	4.35	5.51
1024	1.77	0.971	0.556	0.675	1.14	3.52	5.3	8.31	9.92
4096	2.35	0.763	0.593	1.35	1.88	5.03	7.55	12.5	15.5
16384	2.59	0.889	0.66	1.78	1.97	5.95	8.52	13.2	15.7

It can be seen that, from ~64 points onwards, pyaudi becomes faster than DACE. Clearly this results are only indicative as a specific computation is selected and different sparsities and computations will result in different speedups. It is nonetheless useful to establish a trend which remains true in general: applicatins where very high derivation orders or multiple pointsexpansion points need to be computed, will benefit from `pyaudi` algorithmic implementations.

Comparison with TaylorModels.jl

We here test the performance of the implementation of Taylor models in `pyaudi` against the Julia package TaylorModels.jl. To perform the comparison we use three functions f, g, h: one univariate, one bivariate and one trivariate defined below. We then construct Taylor models of all the variables separately and time the evaluation of the corresponding Taylor model. The comparison is made on a single CPU machine.

$$
\begin{aligned}
f(x, y, z) = & \frac{4\tan(3y)}{3x + x\sqrt{\frac{6x}{-7(x-8)}}} - 120 - 2x - 7z(1+2y) \\
& - \sinh\left(0.5 + \frac{6y}{8y+7}\right) + \frac{(3y+13)^2}{3z} - 20z(2z-5) \\
& + \frac{5x\tanh(0.9z)}{\sqrt{5y}} - 20y\sin(3z) \\
g(x, y) = & \sin(1.7x + 0.5)(y+2)\sin(1.5y) \\
h(x) = & x(x-1.1)(x+2)(x+2.2)(x+2.5)(x+3)\sin(1.7x+0.5)
\end{aligned}
$$

Dimension	Package	Remainder Bound (Order 1)	Remainder Bound (Order 15)	Speed Comparison
h(x)	TaylorModels.jl	1e+2	1e-5	~1–1.5× faster than pyaudi
	pyaudi	1e+2	1e-5	~1–1.5× slower than TaylorModels.jl
g(x, y)	TaylorModels.jl	1e+1	1e-6	Slower: pyaudi is 5× faster (order 3), 15× faster (order 15), 7800× faster (order 1, edge case)
	pyaudi	1e+1	1e-6	Faster (see above)
f(x, y, z)	TaylorModels.jl	1e+0	1e-11	Slower: pyaudi is 8× faster (order 3), 155× faster (order 15), 13000× faster (order 1, edge case)
	pyaudi	1e-1	1e-17	Faster (see above)

In the table above, a clear trend can be seen both in terms of speed and accuracy. For univariate Taylor models, TaylorModels.jl and `pyaudi` have similar performances. At two dimensions, while the remainder bounds are comparable in size, pyaudi is significantly faster, with the speedup increasing with the order of the polynomial. At three dimensions, pyaudi produces significantly tighter bounds and is again significantly faster, with the speedup increasing with the order of the polynomial.

References

A number of references to relevant work and algorithms implemented in pyaudi are:

- (Biscani, 2020)
- (Makino, 1998)
- (Titi & Garloff, 2019)

Other software packages that do similar things are:

- JAX (Bradbury et al., 2018)
- TensorFlow (Abadi et al., 2015)
- PyTorch (Paszke et al., 2019)
- COSY INFINITY (Makino & Berz, 2006)
- DACE (Massari et al., 2018)
- TaylorSeries.jl/TaylorModels.jl (Benet et al., 2019)
- CORA (Althoff, 2015)

Ongoing research

- EclipseNET (Acciarini, Biscani, et al., 2024) (Acciarini et al., 2025)
- CR3BP stochastic continuation (Acciarini, Baresi, et al., 2024)
- Long-term propagation (Caleb & Lizy-Destrez, 2020)
- Rapid nonlinear convex guidance (Burnett & Topputo, 2025)
- Differentiable genetic programming (Izzo et al., 2017)

Acknowledgement of financial support

No financial support was provided for the development of this software.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., … Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems*. https://www.tensorflow.org/

Acciarini, G., Baresi, N., Lloyd, D. J., & Izzo, D. (2024). Stochastic continuation of trajectories in the circular restricted three-body problem via differential algebra. *arXiv Preprint arXiv:2405.18909*.

Acciarini, G., Biscani, F., & Izzo, D. (2024). EclipseNETs: A differentiable description of irregular eclipse conditions. *arXiv Preprint arXiv:2408.05387*.

Acciarini, G., Izzo, D., & Biscani, F. (2025). EclipseNETs: Learning irregular small celestial body silhouettes. *arXiv Preprint arXiv:2504.04455*.

Althoff, M. (2015). An introduction to CORA 2015. *Proc. Of the 1st and 2nd Workshop on Applied Verification for Continuous and Hybrid Systems*, 120–151. https://doi.org/10.29007/zbkv

Benet, L., Forets, M., Sanders, D., & Schilling, C. (2019). TaylorModels. Jl: Taylor models in julia and their application to validated solutions of ODEs. In *SWIM* (pp. 15–16).

Berz, M., Makino, K., & Wan, W. (2014). *An introduction to beam physics*. Taylor & Francis.

Biscani, F. (2020). *Obake: A c++17 library for the symbolic manipulation of sparse polynomials & co.* (Version 0.9.0). https://github.com/bluescarni/obake

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/jax-ml/jax

Burnett, E. R., & Topputo, F. (2025). Rapid nonlinear convex guidance using a monomial method. *Journal of Guidance, Control, and Dynamics*, *48*(4), 736–756.

Caleb, T., & Lizy-Destrez, S. (2020). Can uncertainty propagation solve the mysterious case of snoopy? *International Conference on Uncertainty Quantification & Optimisation*, 109–128.

Izzo, D., Biscani, F., & Mereta, A. (2017). Differentiable genetic programming. *European Conference on Genetic Programming*, 35–51.

Makino, K. (1998). *Rigorous analysis of nonlinear motion in particle accelerators* [PhD thesis]. Michigan State University.

Makino, K., & Berz, M. (2006). Cosy infinity version 9. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, *558*(1), 346–350.

Massari, M., Di Lizia, P., Cavenago, F., & Wittig, A. (2018). Differential algebra software library with automatic code generation for space embedded applications. In *2018 AIAA information systems-AIAA infotech@ aerospace* (p. 0398).

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., & others. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, *32*.

Titi, J., & Garloff, J. (2019). Matrix methods for the tensorial bernstein form. *Applied Mathematics and Computation*, *346*, 254–271.