

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

Dario Izzo<sup>1</sup>, Francesco Biscani<sup>2</sup>, and Sean Cowan<sup>1</sup>

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

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Published: unpublished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

pyaudi is a Python toolbox developed at the [European Space Agency](#) that implements the algebra of truncated Taylor polynomials to achieve high-order order, forward mode, automatic differentiation in a multivariate setting. This form of forward mode automatic differentiation is implemented via C++ class templates exposed to python using pybind11. This allows the generalized dual number type to behave like a drop-in replacement for floats (or other scalar types), while operator overloading propagates derivatives automatically.

All standard mathematical functions are implemented exploiting the nil-potency property of exponentiation in the algebra of truncated Taylor polynomials.

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 as well as bound precisely the truncation error introduced using its corresponding Taylor model. The resulting representations of program outputs can be used to perform fast Monte Carlo simulations, rigorous uncertainty analyses, local inversions of output-input relations as well as high-order sensitivity analysis. The package implements the approach to high-order automated differentiation perfected by Berz and Makino ([Berz et al., 2014](#)), ([Makino, 1998](#)) while introducing original implementation details aimed at increased efficiency in the polynomial multiplication routines and bounding of Taylor models.

The C++/C library DACE ([Massari et al., 2018](#)) also offers an implementation of the differential algebra of truncated Taylor polynomials, (same as pyaudi but with the addition of extra operators completing the algebra into a differential algebra). As opposed to pyaudi, DACE makes use of a polynomial multiplication routine relying on static memory allocations for the storage of monomial coefficients. As outlined in the comparison reported, the difference makes DACE more efficient for single evaluations, an advantage that is lost when evaluating in batches.

The Julia library TaylorSeries.jl/TaylorModels.jl ([Benet et al., 2019](#)) implements Taylor models to compute guaranteed bounds on generic Taylor series, the approach used is very different

from what implemented in `pyaudi` and a preliminary comparison, reported below, shows how it can be greatly outpermed by `pyaudi`.

## Key aspects

The main features of `pyaudi` are:

- Efficient truncated polynomial arithmetic in arbitrary dimensions, built on top of [Obake](#), a C++ library for symbolic manipulation of sparse multivariate polynomials, truncated power series, and Poisson series. Unlike other packages, which often face severe memory bottlenecks as the polynomial order or number of variables grows, `pyaudi` avoids large static memory allocations and keeps computations memory-efficient, at the cost of some extra bookkeeping.
- Vectorized generalized dual numbers, enabling simultaneous evaluation of identical computational graphs at multiple expansion points. This makes it possible to compute high-order tensors efficiently while amortizing the overhead of the extra bookkeeping introduced by the use of [Obake](#).
- Taylor models implemented using Bernstein polynomials for bounding the range of multivariate polynomials. A comparable open-source package, called `TaylorModels.jl`, calculates bounds using Horner's scheme combined with interval arithmetic. A quick test in the next section shows the significant speedup introduced using `pyaudi` for even a relatively simple trivariate polynomial.
- Map inversion algorithm, implementing the algorithm described in ([Berz et al., 2014](#)), thus allowing local inversion of input–output relations of generic computational graphs.

## Comparison with DACE

### Comparison with DACE

For the comparison with DACE, we take two cases: One with simple multiplication between Taylor polynomials. One with vectorization enabled (in `audi` since it does not exist in DACE).

### Multiplication

We use two tenth-order polynomials of the form:

$$p1 = (1 + x1 + x2 + \dots + xn)^{10}$$

$$p2 = (1 - x1 - x2 - \dots - xn)^{10}$$

where  $x1$  etc. are variables. These polynomials are then multiplied and timed. The speed up of `pyaudi` w.r.t. DACE is given below in seconds. It can be seen that `pyaudi` is faster from  $nvars \geq 10$  and order  $\geq 11$ .

nvars↓ Order→	6	7	8	9	10	11	12	13	14	15
6	0.0901	0.115	0.0978	0.125	0.166	0.264	0.369	0.426	0.966	1.04
8	0.0791	0.226	0.166	0.475	0.635	0.977	1.42	1.72	1.98	2.14
10	0.114	0.177	0.631	0.692	0.866	1.36	1.43	1.12	1.70	3.08
12	0.138	0.422	0.544	0.723	0.914	1.32	1.73	2.18	2.84	4.08

## 69 Vectorization-enabled multiplication

70 To showcase the vectorization, we multiply the following polynomial (where each variable has  
71 a variable number of coefficients) five times with itself and time the operation:

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

72 where  $c_v$  are coefficients. The results are displayed in three tables per number of variables  
73 below. It can be seen that, from 64/256 points onwards, pyaudi is significantly faster than  
74 DACE. It should be noted that this test is only indicative due to the limited study using  
75 an arbitrary choice of a polynomial both for the single-thread multiplication as well as the  
76 vectorized version.

### 77 2 variables

points↓ Order →	1	2	3	4	5	6	7	8	9
16	0.122	0.349	0.387	0.395	0.553	0.607	0.535	0.494	0.527
64	1.54	1.42	1.17	1.10	1.52	1.80	1.50	1.42	0.637
256	5.64	4.50	3.77	1.25	1.61	2.14	1.81	1.73	1.66
1024	12.7	3.30	2.68	2.04	2.41	3.11	2.77	2.70	2.73
4096	3.62	3.58	3.28	3.43	3.80	5.30	4.73	4.76	4.35
16384	5.62	4.44	3.48	2.98	3.68	3.91	3.74	3.61	3.33

### 78 5 variables

points↓ Order →	1	2	3	4	5	6	7	8	9
16	0.339	0.579	0.437	0.478	0.219	0.365	0.323	0.374	0.443
64	1.42	1.87	0.591	0.470	0.507	0.799	0.703	0.921	1.11
256	5.27	2.24	1.31	1.19	1.26	2.24	2.27	2.07	2.60
1024	5.23	4.54	2.79	2.31	2.44	3.28	3.19	2.85	3.14
4096	8.10	7.26	4.73	2.61	2.50	3.69	3.38	3.53	3.50
16384	7.20	4.65	3.01	2.32	2.06	3.34	3.33	3.46	3.54

### 79 10 variables

points↓ Order →	1	2	3	4	5	6	7	8	9
16	0.287	0.356	0.158	0.144	0.241	0.539	0.827	1.04	1.50
64	1.19	0.461	0.394	0.387	0.594	1.20	1.55	1.97	2.89
256	4.26	1.50	1.14	1.14	1.45	2.73	3.46	4.44	5.95
1024	4.04	3.41	2.24	1.80	2.10	3.92	4.96	6.45	9.90
4096	8.83	5.38	2.52	2.14	2.53	5.05	5.77	7.41	12.7
16384	3.81	2.33	2.04	2.06	2.61	4.57	5.13	6.78	10.1

## 80 Comparison with TaylorModels.jl

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

f(x, y, z) = (4 tan(3y) / (3x + x \* sqrt(6x / (-7(x-8)))) - 120 - 2x - 7z(1 + 2y) - sinh(0.5 + 6y / (8y + 7)) + ((3y + 13)^2 / 3z) - 20z(2z - 5) + (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)

Table with 5 columns: Dimension, Package, Remainder Bound (Order 1), Remainder Bound (Order 15), and Speed Comparison. It compares TaylorModels.jl and pyaudi for functions h(x), g(x, y), and f(x, y, z) at different polynomial orders.

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. JI: 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*

- 147      *Equipment*, 558(1), 346–350.
- 148      Massari, M., Di Lizia, P., Cavenago, F., & Wittig, A. (2018). Differential algebra software  
149      library with automatic code generation for space embedded applications. In *2018 AIAA*  
150      *information systems-AIAA infotech@ aerospace* (p. 0398).
- 151      Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin,  
152      Z., Gimelshein, N., Antiga, L., & others. (2019). Pytorch: An imperative style, high-  
153      performance deep learning library. *Advances in Neural Information Processing Systems*,  
154      32.
- 155      Titi, J., & Garloff, J. (2019). Matrix methods for the tensorial bernstein form. *Applied*  
156      *Mathematics and Computation*, 346, 254–271.

DRAFT