The Journal of Open Source Software

multivar_horner: A Python package for computing Horner factorisations of multivariate polynomials

Jannik Michelfeit[1, 2]

1 Technische Universität Dresden, Dresden, Germany **2** Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany

DOI: 10.21105/joss.02392

Software

- Review
- Repository
- Archive

Editor: David P. Sanders

Reviewers:

- @henrik227
- @saschatimme

Submitted: 20 May 2020
Published: 02 October 2020

Abstract

Many applications in the sciences require numerically stable and computationally efficient evaluation of multivariate polynomials. Finding beneficial representations of polynomials, such as Horner factorisations, is therefore crucial. `multivar_horner` (Michelfeit, 2018), the Python package presented here, is, as far as we are aware, the first open-source software for computing multivariate Horner factorisations. This paper briefly outlines the functionality of the package and places it in context with respect to previous work in the field. Benchmarks additionally demonstrate the advantages of the implementation and Horner factorisations in general.

Introduction

Polynomials are a central concept in mathematics and find application in a wide range of fields (Akritas, 1989; Boumova, 2002; Cools, 2002; Hecht et al., 2018; Prasolov, 2009). (Multivariate) polynomials have different possible mathematical representations and the beneficial properties of some representations are in great demand in many applications (Hecht et al., 2018; Lee, 2013; Leiserson, Li, Maza, & Xie, 2010).

The *Horner factorisation* is such a representation with beneficial properties. Compared to the unfactorised representation of a multivariate polynomial, in the following called *canonical form*, this representation offers some important advantages. Firstly, the Horner factorisation is more compact, in the sense that it requires fewer mathematical operations in order to evaluate the polynomial (cf. Figure 4). Consequently, evaluating a multivariate polynomial in Horner factorisation is faster and numerically more stable (Ceberio & Kreinovich, 2004; Peña & Sauer, 2000a, 2000b) (cf. Figure 2). These advantages come at the cost of an initial computational effort required to find the factorisation.

The `multivar_horner` Python package implements a multivariate Horner scheme ("Horner's method", "Horner's rule") (Horner, 1819) to compute Horner factorisations of multivariate polynomials given in canonical form. The package offers the functionality of representing multivariate polynomials of arbitrary degree in Horner factorisation as well as in canonical form. Additionally it allows to compute the partial derivatives of a polynomial and to evaluate a polynomial at a given point. Accordingly the package presented here is useful whenever (multivariate) polynomials have to be evaluated efficiently, the numerical error of the polynomial evaluation has to be small, or a compact representation of the polynomial is required. This holds true for many applications applying numerical analysis. One example use case where this package is already being employed are novel response surface methods (Michelfeit, 2019) based on multivariate Newton interpolation (Hecht et al., 2018).

Functionality

`multivar_horner` implements a multivariate Horner scheme using the greedy heuristic presented in (Ceberio & Kreinovich, 2004). In the following the key functionality of this package is outlined. For more details on polynomials and Horner factorisations please refer to the literature, e.g. (Neumaier, 2001).

A polynomial in canonical form is a sum of monomials. For a univariate polynomial $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d$ (canonical form), the Horner factorisation is unique: $f(x) = a_0 + x(a_1 + x(\cdots x(a_d)\cdots)$. In the multivariate case, however, the factorisation is ambiguous, as there are multiple possible factors to factorise with. The key functionality of `multivar_horner` is finding a good instance among the many possible Horner factorisations of a multivariate polynomial.

Let's consider the example multivariate polynomial given in canonical form by $p(x) = 5 + x_1^3 x_2 + 2x_1^2 x_3 + 3x_1 x_2 x_3$. The polynomial p is the sum of 4 monomials, has dimensionality 3 and can also be written as $p(x) = 5x_1^0 x_2^0 x_3^0 + 1x_1^3 x_2^1 x_3^0 + 2x_1^2 x_2^0 x_3^1 + 3x_1^1 x_2^1 x_3^1$. The coefficients of the monomials are 5, 1, 2 and 3 respectively.

From this formulation it is straightforward to represent a multivariate polynomial with a single vector of coefficients and one exponent matrix. Due to its simplicity and universality this kind of representation is used for defining polynomials as input. It should be noted that this most trivial representation is computationally expensive to evaluate.

The number of additions of a polynomial remains constant irrespective of the polynomial factorisation, since it depends solely on the number of monomials and a factorisation does not influence the number of monomials. This holds true only without taking common subexpression elimination into account. Hence the number of additions is irrelevant for evaluating the quality of a factorisation. In the following we accordingly only count the number of multiplications for a less biased comparison to other polynomial representations. Note that each exponentiation is counted as (exponent - 1) operations.

Usage

The following code snippet shows how to use `multivar_horner` to compute a Horner factorisation of p:

```
from multivar_horner import HornerMultivarPolynomial
coefficients = [5.0, 1.0, 2.0, 3.0]
exponents = [[0, 0, 0], [3, 1, 0], [2, 0, 1], [1, 1, 1]]
p = HornerMultivarPolynomial(coefficients, exponents, rectify_input=True,
    compute_representation=True)
```

The factorisation computed by `multivar_horner` is $p(x) = x_1(x_1(x_1(1x_2)+2x_3)+3x_2x_3)+5$ and requires 7 multiplications for every polynomial evaluation. The human readable representation of the polynomial can be accessed with:

```
print(p.representation)
# [#ops=7] p(x) = x_1 (x_1 (x_1 (1.0 x_2) + 2.0 x_3) + 3.0 x_2 x_3) + 5.0
```

It should be noted that the implemented factorisation procedure is coefficient-agnostic and hence does not, for example, optimise multiplications with 1. This design choice has been made in order to have the ability to change the coefficients of a computed polynomial representation a posteriori.

With the default settings a Horner factorisation is computed by recursively factorising with respect to the factor most commonly used in all monomials. When no leaves of the resulting binary "Horner factorisation tree" can be factorised any more, a "recipe" for evaluating the polynomial is compiled. This recipe encodes all operations required to evaluate the polynomial in `numpy` arrays (Walt, Colbert, & Varoquaux, 2011). This enables the use of functions just-in-time compiled by `numba` (Lam, Pitrou, & Seibert, 2015), which allow the polynomial evaluation to be computationally efficient. The just-in-time compiled functions are always used, since a pure-Python polynomial evaluation would to some extent outweigh the benefits of Horner factorisation representations.

`multivar_horner` allows to evaluate the polynomial p at a point x:

```
x = [-2.0, 3.0, 1.0]
p_x = p.eval(x, rectify_input=True) # -29.0
```

Degrees of multivariate polynomials

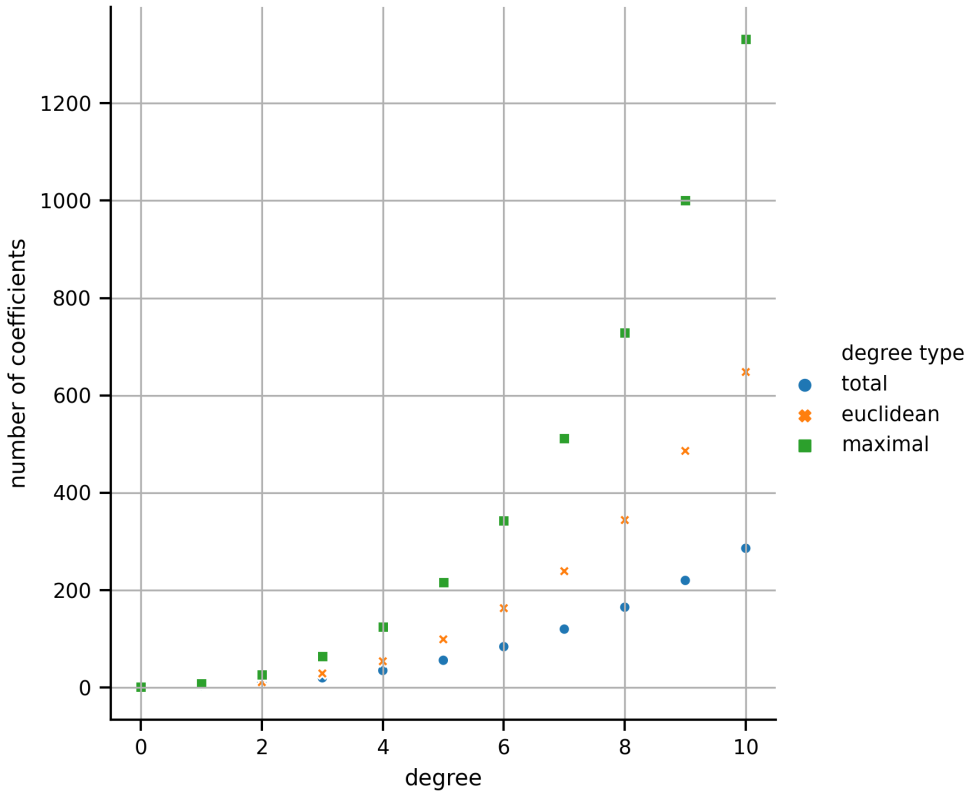

Figure 1: The number of coefficients of fully occupied polynomials of different degrees in 3 dimensions.

In contrast to the one-dimensional case, there are several concepts of degree for polynomials in multiple dimensions. Following the notation of (Trefethen, 2017) the usual notion of degree of a polynomial, the *total degree*, is the maximal sum of exponents of all monomials. This is equal to the maximal l_1-norm of all exponent vectors of the monomials. Accordingly the *euclidean degree* is the maximal l_2-norm and the *maximal degree* is the maximal l_∞-norm of all exponent vectors. Please refer to (Trefethen, 2017) for precise definitions.

A polynomial is called *fully occupied* with respect to a certain degree if all possible monomials having a smaller or equal degree are present. The occupancy of a polynomial can then be defined as the number of monomials present, relative to the fully-occupied polynomial of this degree. A fully-occupied polynomial hence has an occupancy of 1.

The number of coefficients (equal to the number of possible monomials) in multiple dimensions highly depends on the type of degree a polynomial has (cf. Figure 1). This effect intensifies as the dimensionality grows.

Benchmarks

Figure 2: Numerical error of evaluating randomly-generated polynomials of varying sizes.

For benchmarking our method the following procedure is used: In order to sample polynomials with uniformly random occupancy, the probability of monomials being present is picked randomly. For a fixed *maximal* degree n in m dimensions there are $(n+1)^m$ possible exponent vectors corresponding to monomials. Each of these monomials is included with the chosen probability.

Five polynomials were sampled randomly for each maximal degree up to 7 and dimensionality up to 7. In order to compute the numerical error, each polynomial is evaluated at a point sampled uniformly from $[-1;1]^m$ with the different methods. The polynomial evaluation algorithms use 64-bit floating point numbers, whereas the ground truth is computed with 128-bit accuracy in order to avoid numerical errors in the ground truth value. To obtain more representative results, the numerical error is averaged over 100 runs with uniformly-random coefficients each in the range $[-1;1]$. All errors are displayed relative to the ground truth.

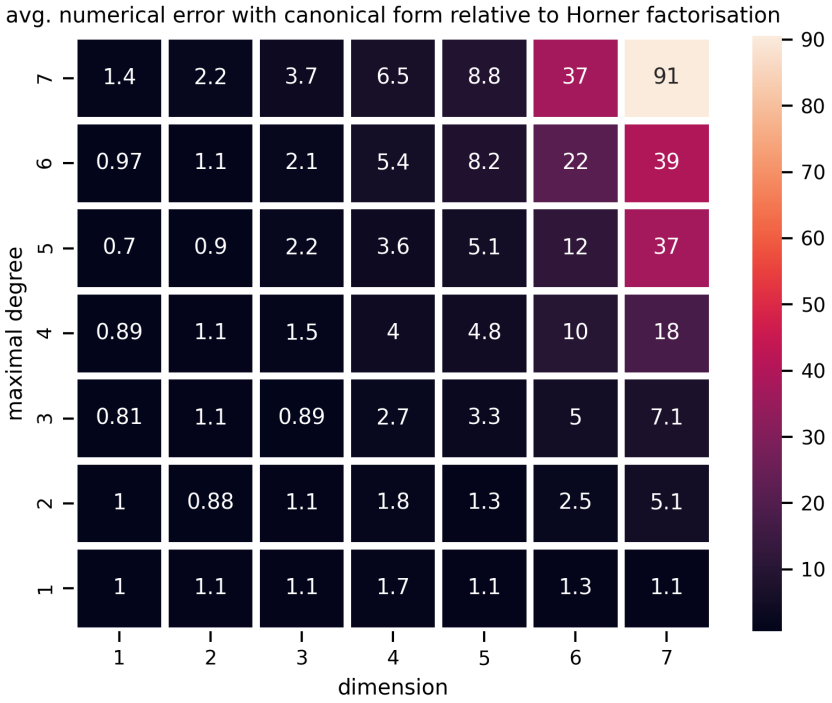

Figure 3: Numerical error of evaluating randomly generated polynomials in canonical form relative to the Horner factorisation.

Note that even though the original monomials are not actually present in a Horner factorisation, the number of coefficients is nonetheless identical to the number of coefficients of its canonical form. With increasing size in terms of the number of included coefficients, the numerical error of both the canonical form and the Horner factorisation found by `multivar_horner` grow exponentially (cf. Figure 2). However, in comparison to the canonical form, the Horner factorisation is more numerically stable, as visualised in Figure 3. The numerical stability of Horner factorisations has theoretically been shown in (Ceberio & Kreinovich, 2004; Peña & Sauer, 2000a, 2000b).

Even though the number of operations required for evaluating the polynomials grows exponentially with their size, irrespective of the considered representation, Figure 4 shows that the rate of growth is lower for the Horner factorisation. As a result, the Horner factorisations are computationally easier to evaluate.

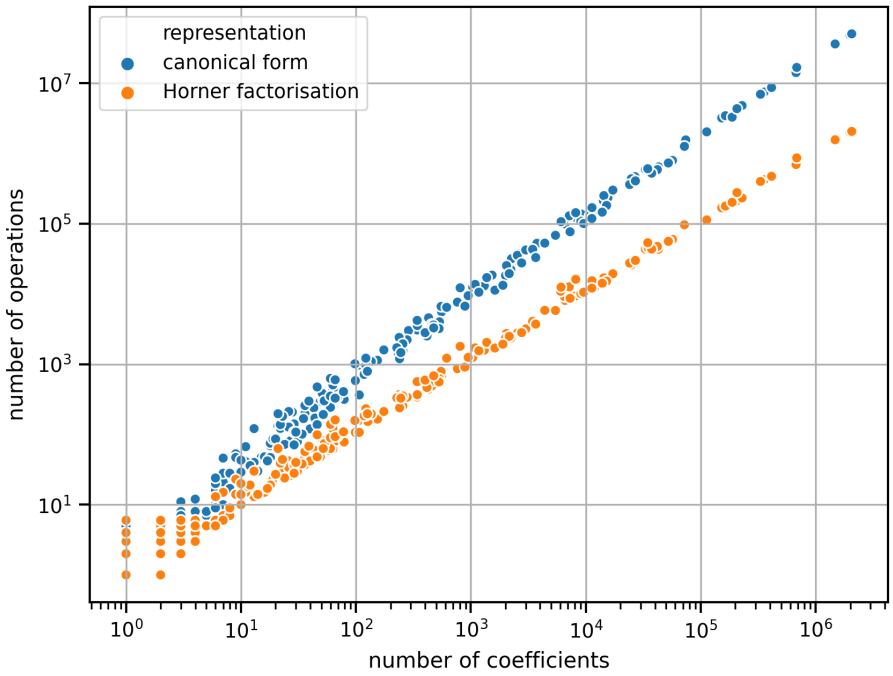

Figure 4: Number of operations required to evaluate randomly generated polynomials.

Related work

This package has been created due to the recent advances in multivariate polynomial interpolation (Hecht et al., 2018; Hecht & Sbalzarini, 2018). High-dimensional interpolants of large degrees create the demand for evaluating multivariate polynomials in a computationally efficient and numerically stable way. These advances enable modeling the behaviour of (physical) systems with polynomials. Obtaining an analytical, multidimensional and nonlinear representation of a system opens up many possibilities. With so-called "interpolation response surface methods" (Michelfeit, 2019), for example, a system can be analysed and optimised.

The commercial software `Maple` offers the ability to compute multivariate Horner factorisations. However `multivar_horner` is, as far as we are aware, the first open-source implementation of a multivariate Horner scheme. The `Wolfram Mathematica` framework supports univariate Horner factorisations. The `Julia` package StaticPolynomials has a functionality similar to `multivar_horner`, but does not support computing Horner factorisations.

`NumPy` (Walt et al., 2011) offers functionality to represent and manipulate polynomials of dimensionality up to 3. `SymPy` offers the dedicated module `sympy.polys` for symbolically operating with polynomials. As stated in the documentation, `SymPy` does not use Horner factorisations for polynomial evaluation in the multivariate case. `Sage` covers the algebraic side of polynomials.

`multivar_horner` has no functions to directly interoperate with other software packages. The generality of the required input parameters (coefficients and exponents), however, still ensures the compatibility with other approaches. It is, for example, easy to manipulate a polynomial with other libraries and then compute the Horner factorisation representation of the resulting output polynomial with `multivar_horner` afterwards, by simply transferring

coefficients and exponents. Some intermediate operations to convert the parameters into the required format might be necessary.

Further reading

The documentation of the package is hosted on readthedocs.io. Any bugs or feature requests can be filed on GitHub (Michelfeit, 2018). The contribution guidelines can be found there as well.

The underlying basic mathematical concepts are explained in numerical analysis textbooks like (Neumaier, 2001). The Horner scheme at the core of `multivar_horner` has been theoretically outlined in (Ceberio & Kreinovich, 2004).

Instead of using a heuristic to choose the next factor, one could instead search over all possible Horner factorisations in order to arrive at a minimal factorisation. The number of possible factorisations, however, increases exponentially with the degree and dimensionality of a polynomial (number of monomials). One possibility to avoid computing each factorisation is to employ a version of A-star search (Hart, Nilsson, & Raphael, 1968) adapted for factorisation trees. `multivar_horner` also implements this approach, which is similar to the branch-and-bound method suggested in (Kojima, 2008, ch. 3.1).

(Carnicer & Gasca, 1990) shows how factorisation trees can be used to evaluate multivariate polynomials and their derivatives. In (Kuipers, Plaat, Vermaseren, & Herik, 2013) Monte Carlo tree search has been used to find more performant factorisations than with greedy heuristics. Other beneficial representations of polynomials are specified, for example, in (Lee, 2013) and (Leiserson et al., 2010).

Acknowledgements

Thanks to Michael Hecht (Max Planck Institute of Molecular Cell Biology and Genetics) and Steve Schmerler (Helmholtz-Zentrum Dresden-Rossendorf) for valuable input enabling this publication. I also thank the editor David P. Sanders (Universidad Nacional Autónoma de México) as well as the reviewers Henrik Barthels (RWTH Aachen University) and Sascha Timme (TU Berlin) for their helpful feedback.

References

Akritas, A. (1989). *Elements of computer algebra with applications*. John Wiley & Sons, Inc.

Boumova, S. (2002). *Applications of polynomials to spherical codes and designs* (PhD thesis). Technische Universiteit Eindhoven.

Carnicer, J., & Gasca, M. (1990). Evaluation of multivariate polynomials and their derivatives. *Mathematics of Computation*, *54*(189), 231–243. doi:10.1090/s0025-5718-1990-0993925-1

Ceberio, M., & Kreinovich, V. (2004). Greedy algorithms for optimizing multivariate Horner schemes. doi:10.1145/980175.980179

Cools, R. (2002). Advances in multidimensional integration. *Journal of computational and applied mathematics*, *149*(1), 1–12. doi:10.1016/s0377-0427(02)00517-4

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Hecht, M., Hoffmann, K. B., Cheeseman, B. L., & Sbalzarini, I. F. (2018). Multivariate Newton interpolation. *arXiv preprint arXiv:1812.04256*.

Hecht, M., & Sbalzarini, I. F. (2018). Fast interpolation and Fourier transform in high-dimensional spaces. In K. Arai, S. Kapoor, & R. Bhatia (Eds.), *Intelligent computing. Proc. 2018 ieee computing conf., vol. 2,* Advances in intelligent systems and computing (Vol. 857, pp. 53–75). London, UK: Springer Nature. doi:10.1007/978-3-030-01177-2_5

Horner, W. G. (1819). XXI. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, (109), 308–335. doi:10.1098/rstl.1819.0023

Kojima, M. (2008). Efficient evaluation of polynomials and their partial derivatives in homotopy continuation methods. *Journal of the Operations Research Society of Japan*, *51*(1), 29–54. doi:10.15807/jorsj.51.29

Kuipers, J., Plaat, A., Vermaseren, J., & Herik, H. J. van den. (2013). Improving multivariate Horner schemes with Monte Carlo tree search. *Computer Physics Communications*, *184*(11), 2391–2395. doi:10.1016/j.cpc.2013.05.008

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A llvm-based python jit compiler. In *Proceedings of the second workshop on the llvm compiler infrastructure in hpc*, LLVM '15. New York, NY, USA: Association for Computing Machinery. doi:10.1145/2833157.2833162

Lee, M. M.-D. (2013, June 13). *Factorization of multivariate polynomials* (PhD thesis). Technische Universität Kaiserslautern.

Leiserson, C. E., Li, L., Maza, M. M., & Xie, Y. (2010). Efficient evaluation of large polynomials. In *International congress on mathematical software* (pp. 342–353). Springer. doi:10.1007/978-3-642-15582-6_55

Michelfeit, J. (2018). Retrieved May 28, 2020, from https://github.com/MrMinimal64/multivar_horner

Michelfeit, J. (2019). *A response surface method based on multivariate newton interpolation* (Student Thesis). Retrieved from https://mosaic.mpi-cbg.de/docs/Michelfeit2019.pdf

Neumaier, A. (2001). *Introduction to numerical analysis*. Cambridge University Press. doi:10.1017/CBO9780511612916

Peña, J. M., & Sauer, T. (2000a). On the multivariate Horner scheme. *SIAM journal on numerical analysis*, *37*(4), 1186–1197.

Peña, J. M., & Sauer, T. (2000b). On the multivariate Horner scheme II: Running error analysis. *Computing*, *65*(4), 313–322. doi:10.1007/s006070070002

Prasolov, V. V. (2009). *Polynomials* (Vol. 11). Springer Science & Business Media.

Trefethen, L. (2017). Multivariate polynomial approximation in the hypercube. *Proceedings of the American Mathematical Society*, *145*(11), 4837–4844.

Walt, S. van der, Colbert, S. C., & Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, *13*(2), 22–30. doi:10.1109/MCSE.2011.37