multivar_horner

Release /bin/sh: 1: poetry: not found

Jannik Michelfeit

Feb 15, 2023

CONTENTS:

a python package implementing a multivariate Horner scheme ("Horner's method", "Horner's rule")[1] for efficiently evaluating multivariate polynomials.

CHAPTER ONE

GETTING STARTED

1.1 Installation

Installation with pip:

```
pip install multivar_horner
```

For efficiency this package is compiling the instructions required for polynomial evaluation to C by default. If you don't have a C compiler (`gcc` or `cc`) installed you also need to install numba for using an alternative method:

```
pip install multivar_horner[numba]
```

1.2 Basics

Let's consider this example multivariate polynomial:

$p(x) = 5 + 1x_1^3x_2^1 + 2x_1^2x_3^1 + 3x_1^1x_2^1x_3^1$

Which can also be written as:

$p(x) = 5x_1^0x_2^0x_3^0 + 1x_1^3x_2^1x_3^0 + 2x_1^2x_2^0x_3^1 + 3x_1^1x_2^1x_3^1$

A polynomial is a sum of monomials. Our example polynomial has $M = 4$ monomials and dimensionality $N = 3$.

The coefficients of our example polynomial are: 5.0, 1.0, 2.0, 3.0

The exponent vectors of the corresponding monomials are:

- [0, 0, 0]
- [3, 1, 0]
- [2, 0, 1]
- [1, 1, 1]

To represent polynomials this package requires the coefficients and the exponent vectors as input.

This code shows how to compute the Horner factorisation of our example polynomial p and evaluating p at a point x:

```
import numpy as np
from multivar_horner import HornerMultivarPolynomial

coefficients = np.array([[5.0], [1.0], [2.0], [3.0]], dtype=np.float64)  # shape: (M,1)
exponents = np.array(
```

(continues on next page)

(continued from previous page)

```
    [[0, 0, 0], [3, 1, 0], [2, 0, 1], [1, 1, 1]], dtype=np.uint32
)  # shape: (M,N)
p = HornerMultivarPolynomial(coefficients, exponents)

x = np.array([-2.0, 3.0, 1.0], dtype=np.float64)  # shape: (1,N)
p_x = p(x)  # -29.0
```

Note: with the default settings the input is required to have these data types and shapes

With the class `HornerMultivarPolynomial` a polynomial can be represented in *Horner factorisation*.

With the class `HornerMultivarPolynomialOpt` a polynomial can be represented in an *optimal Horner factorisation*.

With the class `MultivarPolynomial` a polynomial can be represented in *canonical form*.

All available features of this package are explained *HERE*.

The API documentation can be found *HERE*.

CHAPTER

TWO

USAGE

Note: Also check out the *API documentation* or the code.

Let's look at the example multivariate polynomial:

$p(x) = 5 + 1x_1^3 x_2^1 + 2x_1^2 x_3^1 + 3x_1^1 x_2^1 x_3^1$

Which can also be written as:

$p(x) = 5x_1^0 x_2^0 x_3^0 + 1x_1^3 x_2^1 x_3^0 + 2x_1^2 x_2^0 x_3^1 + 3x_1^1 x_2^1 x_3^1$

A polynomial is a sum of monomials. Our example polynomial has $M = 4$ monomials and dimensionality $N = 3$.

The coefficients of our example polynomial are: 5.0, 1.0, 2.0, 3.0

The exponent vectors of the corresponding monomials are:

- [0, 0, 0]
- [3, 1, 0]
- [2, 0, 1]
- [1, 1, 1]

To represent polynomials this package requires the coefficients and the exponent vectors as input.

```
import numpy as np

coefficients = np.array(
    [[5.0], [1.0], [2.0], [3.0]], dtype=np.float64
)  # numpy (M,1) ndarray
exponents = np.array(
    [[0, 0, 0], [3, 1, 0], [2, 0, 1], [1, 1, 1]], dtype=np.uint32
)  # numpy (M,N) ndarray
```

Note: by default the Numba jit compiled functions require these data types and shapes

2.1 Horner factorisation

to create a representation of the multivariate polynomial p in Horner factorisation:

```
from multivar_horner import HornerMultivarPolynomial

horner_polynomial = HornerMultivarPolynomial(coefficients, exponents)
```

the found factorisation is $p(x) = x_1^1(x_1^1(x_1^1(1.0x_2^1) + 2.0x_3^1) + 3.0x_2^1x_3^1) + 5.0$.

pass `rectify_input=True` to automatically try converting the input to the required `numpy` data structures and types

```
coefficients = [5.0, 1.0, 2.0, 3.0]
exponents = [[0, 0, 0], [3, 1, 0], [2, 0, 1], [1, 1, 1]]
horner_polynomial = HornerMultivarPolynomial(
    coefficients, exponents, rectify_input=True
)
```

pass `keep_tree=True` during construction of a Horner factorised polynomial, when its factorisation tree should be kept after the factorisation process (e.g. to be able to compute string representations of the polynomials later on)

```
horner_polynomial = HornerMultivarPolynomial(coefficients, exponents, keep_tree=True)
```

Note: for increased efficiency the default for both options is `False`

2.2 canonical form

it is possible to represent the polynomial without any factorisation (refered to as 'canonical form' or 'normal form'):

```
from multivar_horner import MultivarPolynomial

polynomial = MultivarPolynomial(coefficients, exponents)
```

use this if …

- the Horner factorisation takes too long
- the polynomial is going to be evaluated only a few times
- fast polynomial evaluation is not required or
- the numerical stability of the evaluation is not important

Note: in the case of unfactorised polynomials many unnecessary operations are being done (internally uses naive numpy matrix operations)

2.3 string representation

in order to compile a string representation of a polynomial pass `compute_representation=True` during construction

Note: the number in square brackets indicates the number of multiplications required to evaluate the polynomial.

Note: exponentiations are counted as exponent - 1 operations, e.g. x^3 <-> 2 operations

```
polynomial = MultivarPolynomial(coefficients, exponents)
print(polynomial)  # [#ops=10] p(x)


polynomial = MultivarPolynomial(coefficients, exponents, compute_representation=True)
print(polynomial)
# [#ops=10] p(x) = 5.0 x_1^0 x_2^0 x_3^0 + 1.0 x_1^3 x_2^1 x_3^0 + 2.0 x_1^2 x_2^0 x_3^1␣
↪+ 3.0 x_1^1 x_2^1 x_3^1

horner_polynomial = HornerMultivarPolynomial(
    coefficients, exponents, compute_representation=True
)
print(horner_polynomial.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
```

the formatting of the string representation can be changed with the parameters `coeff_fmt_str` and `factor_fmt_str`:

```
polynomial = MultivarPolynomial(
    coefficients,
    exponents,
    compute_representation=True,
    coeff_fmt_str="{:1.1e}",
    factor_fmt_str="(x{dim} ** {exp})",
)
```

the string representation can be computed after construction as well.

Note: for `HornerMultivarPolynomial`: `keep_tree=True` is required at construction time

```
polynomial.compute_string_representation(
    coeff_fmt_str="{:1.1e}", factor_fmt_str="(x{dim} ** {exp})"
)
print(polynomial)
# [#ops=10] p(x) = 5.0e+00 (x1 ** 0) (x2 ** 0) (x3 ** 0) + 1.0e+00 (x1 ** 3) (x2 ** 1)␣
↪(x3 ** 0)
#                + 2.0e+00 (x1 ** 2) (x2 ** 0) (x3 ** 1) + 3.0e+00 (x1 ** 1) (x2 **␣
↪1) (x3 ** 1)
```

2.4 change the coefficients of a polynomial

in order to access the polynomial string representation with the updated coefficients pass `compute_representation=True` with `in_place=False` a new polygon object is being generated

Note: the string representation of a polynomial in Horner factorisation depends on the factorisation tree. the polynomial object must hence have keep_tree=True

```
new_coefficients = [
    7.0,
    2.0,
    0.5,
    0.75,
]  # must not be a ndarray, but the length must still fit
new_polynomial = horner_polynomial.change_coefficients(
    new_coefficients,
    rectify_input=True,
    compute_representation=True,
    in_place=False,
)
```

2.5 optimal Horner factorisations

use the class `HornerMultivarPolynomialOpt` for the construction of the polynomial to trigger an adapted A* search to find the optimal factorisation.

See *this chapter* for further information.

Note: time and memory consumption is MUCH higher!

```
from multivar_horner import HornerMultivarPolynomialOpt

horner_polynomial_optimal = HornerMultivarPolynomialOpt(
    coefficients,
    exponents,
    compute_representation=True,
    rectify_input=True,
)
```

2.6 Caching

by default the instructions required for evaluating a Horner factorised polynomial will be cached either as `.c` file or `.pickle` file in the case of `numpy+numba` evaluation.

One can explicitly force the compilation of the instructions in the required format:

```
horner_polynomial = HornerMultivarPolynomial(
    coefficients, exponents, store_c_instr=True, store_numpy_recipe=True
)
```

If you construct a Horner polynomial with the same properties (= exponents) these cached instructions will be used for evaluation and a factorisation won't be computed again. Note that as a consequence you won't be able to access the factorisation tree and string representation in these cases.

the cached files are being stored in `<path/to/env/>multivar_horner/multivar_horner/__pychache__/`

```
horner_polynomial.c_file
horner_polynomial.c_file_compiled
horner_polynomial.recipe_file
```

you can read the content of the cached C instructions:

```
instr = horner_polynomial.get_c_instructions()
print(instr)
```

you can also export the whole polynomial class (including the string representation etc.):

```
path = "file_name.pickle"
polynomial.export_pickle(path=path)
```

to load again:

```
from multivar_horner import load_pickle

polynomial = load_pickle(path)
```

2.7 evaluating a polynomial

in order to evaluate a polynomial at a point `x`:

```
# define a query point and evaluate the polynomial
x = np.array([-2.0, 3.0, 1.0], dtype=np.float64)  # numpy ndarray with shape [N]
p_x = polynomial(x)  # -29.0
```

or

```
p_x = polynomial.eval(x)  # -29.0
```

or

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

As during construction of a polynomial instance, pass `rectify_input=True` to automatically try converting the input to the required `numpy` data structure.

Note: the default for both options is `False` for increased speed

Note: the dtypes are fixed due to the just in time compiled `Numba` functions

2.8 evaluating a polynomial at complex query points

```
x = [np.complex(-2.0, 1.0), np.complex(3.0, -1.0), np.complex(1.0, 0.5)]
p_x = polynomial.eval_complex(x, rectify_input=True)
```

2.9 computing the partial derivative of a polynomial

Note: BETA: untested feature

Note: partial derivatives will be instances of the same parent class

Note: all given additional arguments will be passed to the constructor of the derivative polynomial

Note: dimension counting starts with 1 -> the first dimension is #1!

```
deriv_2 = polynomial.get_partial_derivative(2, compute_representation=True)
# p(x) = x_1 (x_1^2 (1.0) + 3.0 x_3)
```

2.10 computing the gradient of a polynomial

Note: BETA: untested feature

Note: all given additional arguments will be passed to the constructor of the derivative polynomials

```
grad = polynomial.get_gradient(compute_representation=True)
# grad = [
#     p(x) = x_1 (x_1 (3.0 x_2) + 4.0 x_3) + 3.0 x_2 x_3,
```

(continues on next page)

(continued from previous page)

```
#     p(x) = x_1 (x_1^2 (1.0) + 3.0 x_3),
#     p(x) = x_1 (x_1 (2.0) + 3.0 x_2)
# ]
```

CHAPTER

THREE

ABOUT

multivar_horner is a python package implementing a multivariate Horner scheme ("Horner's method", "Horner's rule")[1] for efficiently evaluating multivariate polynomials.

For an explanation of multivariate Horner factorisations and the terminology used here refer to e.g. Greedy algorithms for optimizing multivariate Horner schemes [2]

A given input polynomial in canonical form (or normal form) is being factorised according to the greedy heuristic described in [2] with some additional computational tweaks. The resulting Horner factorisation requires less operations for evaluation and is being computed by growing a "Horner Factorisation Tree". When the polynomial is fully factorized (= all leaves cannot be factorised any more), a computational "recipe" for evaluating the polynomial is being compiled. This "recipe" (stored internally as numpy arrays) enables computationally efficient evaluation. Numba just in time compiled functions operating on the numpy arrays make this fast. All factors appearing in the factorisation are being evaluated only once (reusing computed values).

Pros:

- computationally efficient representation of a multivariate polynomial in the sense of space and time complexity of the evaluation
- less roundoff errors [3, 4]
- lower error propagation, because of fewer operations [2]

Cons:

- increased initial computational requirements and memory to find and then store the factorisation

The impact of computing Horner factorisations has been evaluated in the *benchmarks below*.

With this package it is also possible to represent polynomials in *canonical form* and to search for an *optimal Horner factorisation*.

Also see: GitHub, PyPI, arXiv paper

3.1 Dependencies

python>=3.6, numpy>=1.16, numba>=0.48

3.2 License

multivar_horner is distributed under the terms of the MIT license (see LICENSE).

3.3 Benchmarks

To obtain meaningful results the benchmarks presented here use polynomials sampled randomly with the following procedure: In order to draw 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 being activated with the chosen probability.

Refer to [5] for an exact definition of the maximal degree.

For each maximal degree up to 7 and until dimensionality 7, 5 polynomials were drawn randomly. Note that even though the original monomials are not actually present in a Horner factorisation, the amount of coefficients however is identical to the amount of coefficients of its canonical form.

Even though the amount of operations required for evaluating the polynomials grow exponentially with their size irrespective of the representation, the rate of growth is lower for the Horner factorisation.

Due to this, the bigger the polynomial the more compact the Horner factorisation representation is relative to the canonical form. As a result the Horner factorisations are computationally easier to evaluate.

3.3.1 Numerical error

In order to compute the numerical error, each polynomial has been evaluated at a point chosen uniformly random from $[-1; 1]^m$ with the different methods. The polynomial evaluation algorithms use 64-bit floating point numbers, whereas the ground truth has been computed with 128-bit accuracy in order to avoid numerical errors in the ground truth value. To receive more representative results, the obtained numerical error is being averaged over 100 tries with uniformly random coefficients each in the range $[-1; 1]$, All errors are displayed as (averaged) absolute values.

With increasing size in terms of the amount of included coefficients the numerical error of both the canonical form and the Horner factorisation found by multivar_horner grow exponentially.

In comparison to the canonical form however the Horner factorisation is much more numerically stable. This has also been visualised in the following figure:

Note: if you require an even higher numerical stability you can set FLOAT_DTYPE = numpy.float128 or FLOAT_DTYPE = numpy.longfloat in global_settings.py. Then however the jit compilation has to be removed in helper_fcts_numba.py (Numba does not support float128).

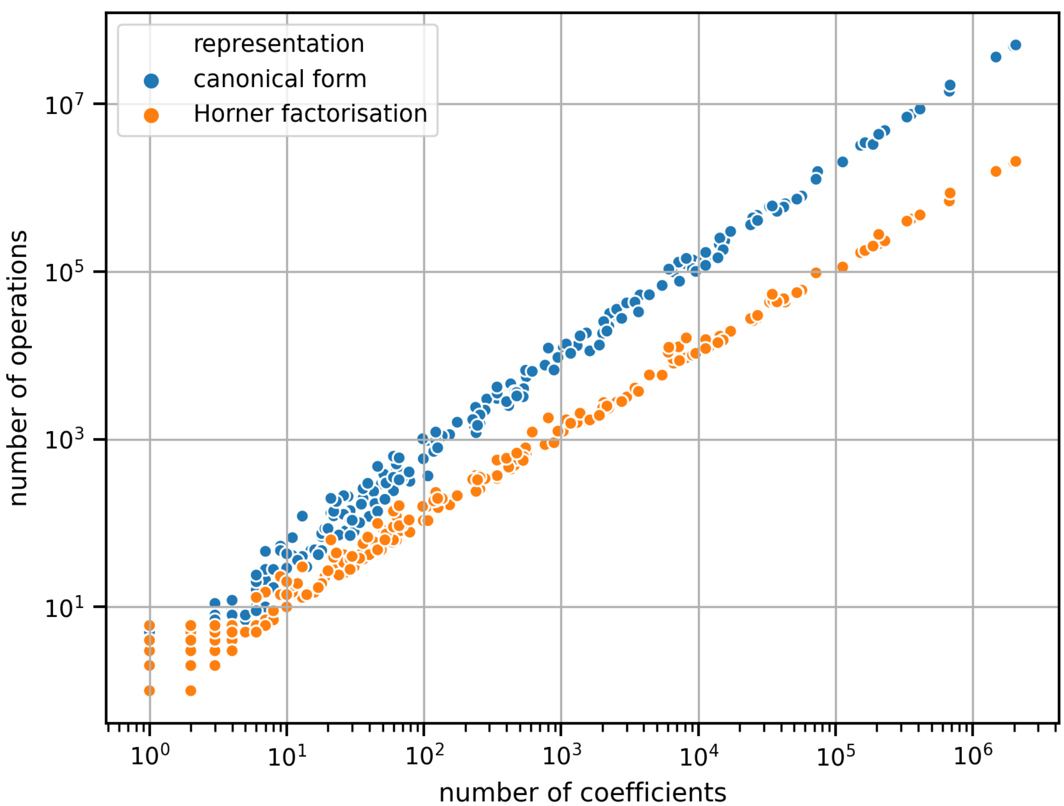

Fig. 1: amount of operations required to evaluate randomly generated polynomials.

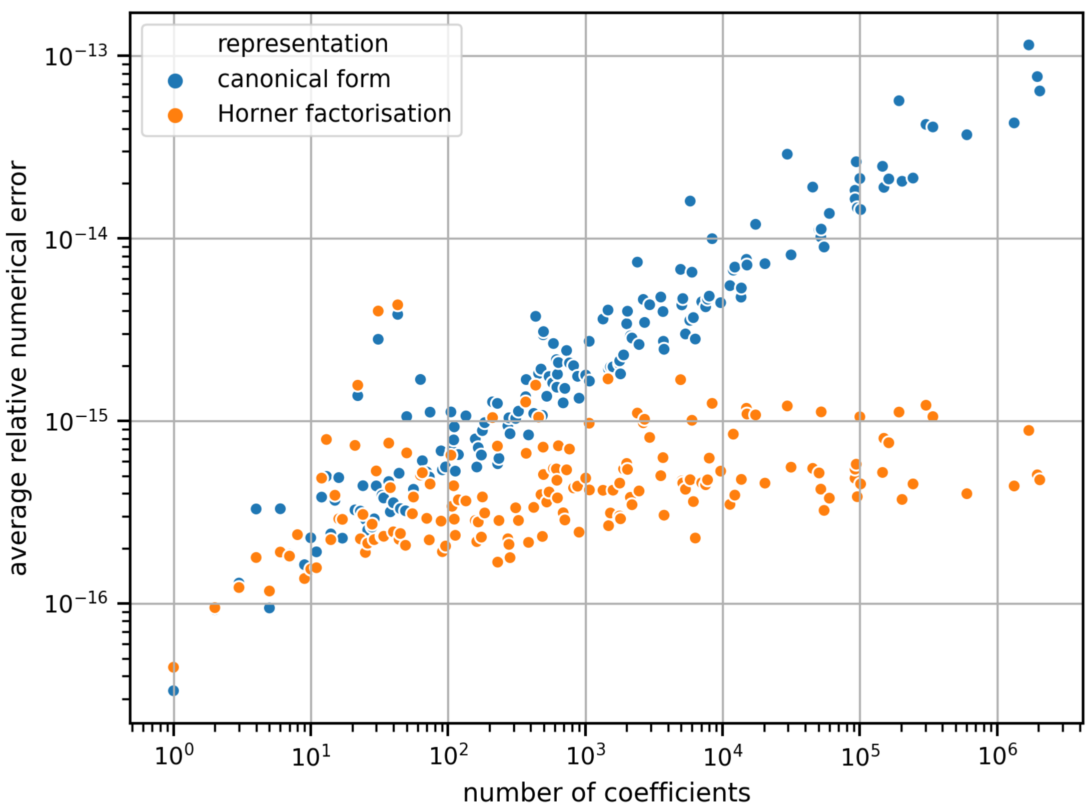

Fig. 2: numerical error of evaluating randomly generated polynomials of varying sizes.

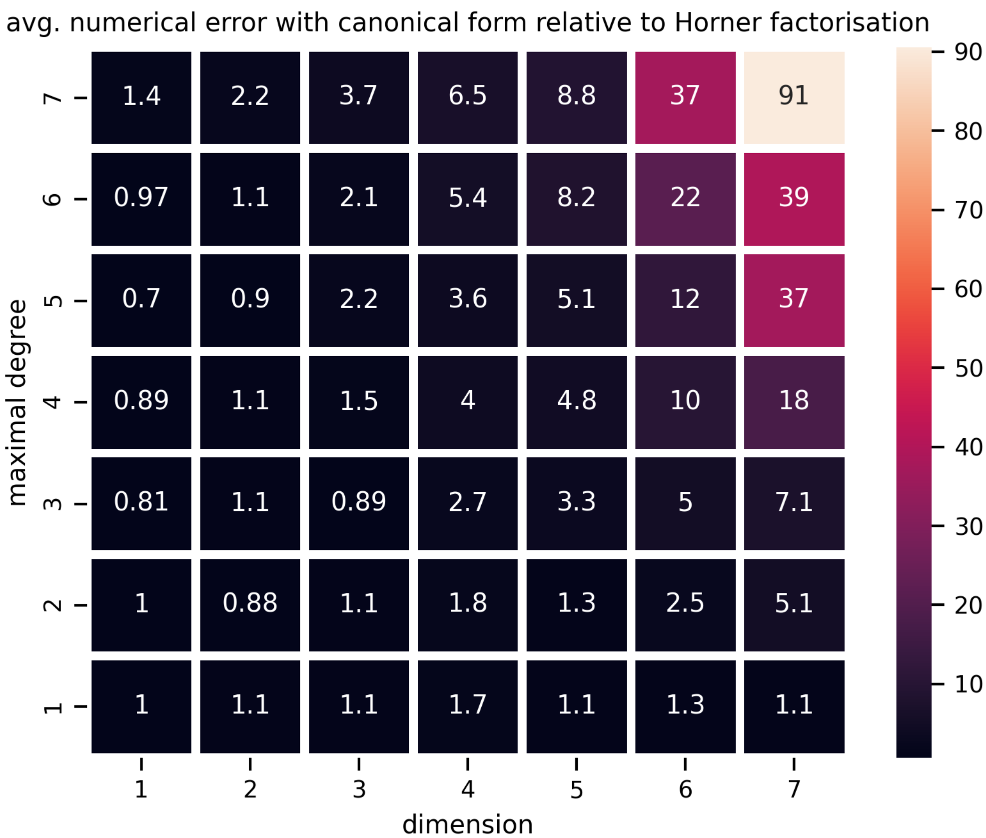

Fig. 3: numerical error of evaluating randomly generated polynomials in canonical form relative to the Horner factorisation.

3.3.2 Speed tests

The following speed benchmarks have been performed on a 2017 MacBook Pro: 4x2,8 GHz Intel Core i7 CPU, 16 GB 2133 MHz LPDDR3 RAM, macOS 10.13 High Sierra. The software versions in use were: `multivar_horner 2.0.0`, `python 3.8.2`, `numpy 1.18.1` and `numba 0.48.0` Both evaluation algorithms (with and without Horner factorisation) make use of `Numba` just in time compiled functions.

```
Speed test:
testing 100 evenly distributed random polynomials
average timings per polynomial:

 parameters   |  setup time (s)                         |  eval time (s)                      ␣
↪      | # operations                        | lucrative after
dim | max_deg | canonical  | horner     | delta           | canonical   | horner       |␣
↪delta        | canonical  | horner     | delta         |    # evals
============================================================================================================
1   | 1       | 4.90e-05   | 2.33e-04   | 3.8 x more      | 8.96e-06    | 1.28e-05     | 0.4␣
↪x more | 3           | 1           | 2.0 x less | -
1   | 2       | 5.24e-05   | 1.95e-04   | 2.7 x more      | 3.42e-06    | 6.01e-06     | 0.8␣
↪x more | 4           | 2           | 1.0 x less | -
1   | 3       | 5.07e-05   | 2.31e-04   | 3.6 x more      | 3.48e-06    | 5.86e-06     | 0.7␣
↪x more | 6           | 3           | 1.0 x less | -
1   | 4       | 5.04e-05   | 2.65e-04   | 4.3 x more      | 3.59e-06    | 5.62e-06     | 0.6␣
↪x more | 7           | 4           | 0.8 x less | -
1   | 5       | 5.08e-05   | 3.04e-04   | 5.0 x more      | 3.49e-06    | 8.47e-06     | 1.4␣
↪x more | 8           | 6           | 0.3 x less | -
1   | 6       | 4.81e-05   | 4.65e-04   | 8.7 x more      | 3.54e-06    | 6.72e-06     | 0.9␣
↪x more | 10          | 7           | 0.4 x less | -
1   | 7       | 5.39e-05   | 4.00e-04   | 6.4 x more      | 3.95e-06    | 6.49e-06     | 0.6␣
↪x more | 12          | 8           | 0.5 x less | -
1   | 8       | 5.19e-05   | 3.83e-04   | 6.4 x more      | 5.63e-06    | 6.16e-06     | 0.1␣
↪x more | 12          | 8           | 0.5 x less | -
1   | 9       | 4.88e-05   | 4.42e-04   | 8.0 x more      | 3.73e-06    | 6.05e-06     | 0.6␣
↪x more | 14          | 10          | 0.4 x less | -
1   | 10      | 4.89e-05   | 5.41e-04   | 10 x more       | 3.80e-06    | 7.11e-06     | 0.9␣
↪x more | 15          | 10          | 0.5 x less | -

2   | 1       | 8.34e-05   | 3.11e-04   | 2.7 x more      | 3.85e-06    | 6.09e-06     | 0.6␣
↪x more | 11          | 3           | 2.7 x less | -
2   | 2       | 4.96e-05   | 7.05e-04   | 13 x more       | 3.80e-06    | 5.82e-06     | 0.5␣
↪x more | 26          | 10          | 1.6 x less | -
2   | 3       | 5.20e-05   | 9.75e-04   | 18 x more       | 4.50e-06    | 6.70e-06     | 0.5␣
↪x more | 38          | 16          | 1.4 x less | -
2   | 4       | 5.93e-05   | 1.44e-03   | 23 x more       | 5.53e-06    | 7.12e-06     | 0.3␣
↪x more | 63          | 27          | 1.3 x less | -
2   | 5       | 5.26e-05   | 2.25e-03   | 42 x more       | 6.49e-06    | 6.46e-06     | -0.0␣
↪x more | 91          | 39          | 1.3 x less | 59828
2   | 6       | 5.31e-05   | 2.90e-03   | 54 x more       | 7.65e-06    | 6.55e-06     | 0.2␣
↪x less | 127         | 54          | 1.4 x less | 2595
2   | 7       | 5.72e-05   | 3.76e-03   | 65 x more       | 9.02e-06    | 6.03e-06     | 0.5␣
↪x less | 164         | 70          | 1.3 x less | 1238
2   | 8       | 5.32e-05   | 4.39e-03   | 81 x more       | 9.71e-06    | 6.06e-06     | 0.6␣
↪x less | 198         | 84          | 1.4 x less | 1186
```

(continues on next page)

(continued from previous page)

```
2    | 9        | 5.27e-05   | 5.04e-03   | 95 x more     | 1.08e-05    | 7.25e-06    | 0.5␣
↪x less | 230          | 99           | 1.3 x less | 1418
2    | 10       | 5.47e-05   | 6.74e-03   | 122 x more    | 1.36e-05    | 6.46e-06    | 1.1␣
↪x less | 310          | 132          | 1.3 x less | 935

3    | 1        | 4.96e-05   | 5.69e-04   | 10 x more     | 3.70e-06    | 6.18e-06    | 0.7␣
↪x more | 26           | 7            | 2.7 x less | -
3    | 2        | 5.34e-05   | 2.02e-03   | 37 x more     | 5.43e-06    | 6.70e-06    | 0.2␣
↪x more | 97           | 28           | 2.5 x less | -
3    | 3        | 5.42e-05   | 4.47e-03   | 82 x more     | 8.88e-06    | 6.13e-06    | 0.4␣
↪x less | 222          | 68           | 2.3 x less | 1605
3    | 4        | 5.59e-05   | 8.40e-03   | 149 x more    | 1.44e-05    | 6.92e-06    | 1.1␣
↪x less | 434          | 133          | 2.3 x less | 1115
3    | 5        | 5.73e-05   | 1.35e-02   | 236 x more    | 2.10e-05    | 1.36e-05    | 0.5␣
↪x less | 685          | 211          | 2.2 x less | 1809
3    | 6        | 7.70e-05   | 2.32e-02   | 300 x more    | 3.72e-05    | 8.75e-06    | 3.3␣
↪x less | 1159         | 355          | 2.3 x less | 811
3    | 7        | 6.86e-05   | 3.46e-02   | 504 x more    | 5.71e-05    | 8.90e-06    | 5.4␣
↪x less | 1787         | 543          | 2.3 x less | 717
3    | 8        | 7.07e-05   | 4.64e-02   | 655 x more    | 6.97e-05    | 9.97e-06    | 6.0␣
↪x less | 2402         | 730          | 2.3 x less | 775
3    | 9        | 8.34e-05   | 6.90e-02   | 826 x more    | 1.05e-04    | 1.15e-05    | 8.2␣
↪x less | 3613         | 1084         | 2.3 x less | 736
3    | 10       | 9.21e-05   | 9.54e-02   | 1034 x more   | 1.42e-04    | 1.35e-05    | 9.5␣
↪x less | 4988         | 1485         | 2.4 x less | 742

4    | 1        | 5.45e-05   | 1.25e-03   | 22 x more     | 4.94e-06    | 6.49e-06    | 0.3␣
↪x more | 67           | 14           | 3.8 x less | -
4    | 2        | 5.83e-05   | 7.20e-03   | 122 x more    | 1.19e-05    | 7.65e-06    | 0.6␣
↪x less | 390          | 91           | 3.3 x less | 1673
4    | 3        | 6.57e-05   | 2.35e-02   | 357 x more    | 3.39e-05    | 7.93e-06    | 3.3␣
↪x less | 1295         | 303          | 3.3 x less | 903
4    | 4        | 7.22e-05   | 4.96e-02   | 686 x more    | 6.68e-05    | 1.02e-05    | 5.6␣
↪x less | 2753         | 653          | 3.2 x less | 874
4    | 5        | 9.85e-05   | 1.17e-01   | 1186 x more   | 1.56e-04    | 1.74e-05    | 8.0␣
↪x less | 6588         | 1535         | 3.3 x less | 843
4    | 6        | 1.40e-04   | 1.98e-01   | 1416 x more   | 2.66e-04    | 1.96e-05    | 13 x␣
↪less   | 11036        | 2582         | 3.3 x less | 802
4    | 7        | 1.77e-04   | 3.27e-01   | 1846 x more   | 4.29e-04    | 2.93e-05    | 14 x␣
↪less   | 18271        | 4276         | 3.3 x less | 820
4    | 8        | 2.77e-04   | 5.97e-01   | 2153 x more   | 8.33e-04    | 4.72e-05    | 17 x␣
↪less   | 33518        | 7736         | 3.3 x less | 760
4    | 9        | 3.82e-04   | 8.90e-01   | 2330 x more   | 1.16e-03    | 6.35e-05    | 17 x␣
↪less   | 47086        | 10944        | 3.3 x less | 812
4    | 10       | 5.44e-04   | 1.30e+00   | 2388 x more   | 1.80e-03    | 8.80e-05    | 20 x␣
↪less   | 73109        | 16873        | 3.3 x less | 758
```

3.4 Related work

This package has been created due to the recent advances in multivariate polynomial interpolation [6, 7]. High dimensional interpolants of large degrees create the demand for evaluating multivariate polynomials computationally efficient and numerically stable.

[8] shows how factorisation trees can be used to evaluate multivariate polynomials and their derivatives.

In [9] Monte Carlo tree search has been used to find more performant factorisations than with greedy heuristics.

Other representations of polynomials are being presented, among others, in [10, 11].

3.5 Contact

Tell me if and how your are using this package. This encourages me to develop and test it further.

Most certainly there is stuff I missed, things I could have optimized even further or explained more clearly, etc. I would be really glad to get some feedback.

If you encounter any bugs, have suggestions etc. do not hesitate to **open an Issue** or **add a Pull Requests** on Git. Please refer to the *contribution guidelines*

3.6 Acknowledgements

Thanks to:

Steve for valuable feedback and writing tests.

CHAPTER

FOUR

OPTIMAL HORNER FACTORISATIONS

See *this code* for an example usage.

Instead of using a heuristic to choose the next factor one can allow a search over all possible (meaningful) factorisations in order to arrive at a minimal Horner factorisation. The amount of possible factorisations however is increasing exponentially with the degree of a polynomial and its amount of monomials. One possibility to avoid computing each factorisation is to employ a version of A*-search [12] adapted for factorisation trees: • Initialise a set of all meaningful possible first level Newton factorisations • Rank all factorisation according to a lower bound ("heuristic") of their lowest possible amount of operations • Iteratively factorise the most promising factorisation and update the heuristic • Stop when the most promising factorisation is fully factorised

This approach is guaranteed to yield a minimal Horner factorisation, but its performance highly depends on the heuristic in use: Irrelevant factorisations are only being ignored if the heuristic is not too optimistic in estimating the amount of operations. On the other hand the heuristic must be easy to compute, because it would otherwise be computationally cheaper to just try all different factorisations. Even though it missing to cover exponentiations, the branch-and-bound method suggested in [13] (ch. 3.1) is almost identical to this procedure.

Even with a good heuristic this method is only traceable for small polynomials because of its increased resource requirements. Since experiments show that factorisations obtained by choosing one factorisation according to a heuristic have the same or only a slightly higher amount of included operations [13] (ch. 7), the computational effort of this approach is not justifiable in most cases. A use case however is to compute and store a minimal representation of a polynomial in advance if possible.

NOTES:

- for the small polynomial examples in the current tests, the results were identical (in terms of #ops) with the approach of just using the default heuristic = trying one factorisation (further analysis needed)!
- in some cases this approach currently is trying all possible factorisations, because the heuristic in use is too optimistic (= brute force, further analysis and improvements needed)
- this requires MUCH more computational resources than just trying one factorisation (the number of possible factorisations is growing exponentially with the size of the polynomial!).
- there are possibly many optimal Horner factorisations of a multivariate polynomial. one could easily adapt this approach to find all optimal Horner factorisations
- even an optimal Horner factorisation must not be the globally minimal representation of a polynomial. there are possibly better types of factorisations and techniques: e.g. "algebraic factorisation", "common subexpression elimination"
- there are multiple possible concepts of optimality (or minimality) of a polynomial

CHAPTER

FIVE

API DOCUMENTATION

5.1 HornerMultivarPolynomial

class multivar_horner.**HornerMultivarPolynomial**(*coefficients*, *exponents*, *rectify_input: bool = False*, *compute_representation: bool = False*, *verbose: bool = False*, *keep_tree: bool = False*, *store_c_instr: bool = False*, *store_numpy_recipe: bool = False*, **args*, ***kwargs*)

Bases: `AbstractPolynomial`

a representation of a multivariate polynomial using Horner factorisation

after computing the factorised representation of the polynomial, the instructions required for evaluation are being compiled and stored. The default format is C instructions. When there is no C compiler installed, the fallback option is encoding the required instructions in numpy arrays (here referred to as "recipes"), which can be processed by (`numba`) just in time compiled functions.

Parameters

- **coefficients** – ndarray of floats with shape (N,1) representing the coefficients of the monomials NOTE: coefficients with value 0 and 1 are allowed and will not affect the internal representation, because coefficients must be replaceable
- **exponents** – ndarray of unsigned integers with shape (N,m) representing the exponents of the monomials where m is the number of dimensions (self.dim), the ordering corresponds to the ordering of the coefficients, every exponent row has to be unique!
- **rectify_input** – bool, default=False whether to convert coefficients and exponents into compatible numpy arrays with this set to True, coefficients and exponents can be given in standard python arrays
- **compute_representation** – bool, default=False whether to compute a string representation of the polynomial
- **verbose** – bool, default=False whether to print status statements
- **keep_tree** – whether the factorisation tree object should be kept in memory after finishing factorisation
- **store_c_instr** – whether a C file with all required evaluation instructions should be created under any circumstances. By default the class will use C based evaluation, but skip if no compiler (`gcc`/`cc`) is installed.
- **store_numpy_recipe** – whether a pickle file with all required evaluation instructions in the custom `numpy+numba` format should be created under any circumstances. By default the class will use C based evaluation and only use this evaluation format as fallback.

num_monomials
the amount of coefficients/monomials N of the polynomial

dim
the dimensionality m of the polynomial NOTE: the polynomial needs not to actually depend on all m dimensions

unused_variables
the dimensions the polynomial does not depend on

num_ops
the amount of mathematical operations required to evaluate the polynomial in this representation

representation
a human readable string visualising the polynomial representation

total_degree
the usual notion of degree for a polynomial. = the maximum sum of exponents in any of its monomials = the maximum l_1-norm of the exponent vectors of all monomials in contrast to 1D polynomials, different concepts of degrees exist for polynomials in multiple dimensions. following the naming in [1] L. Trefethen, "Multivariate polynomial approximation in the hypercube", Proceedings of the American Mathematical Society, vol. 145, no. 11, pp. 4837–4844, 2017.

euclidean_degree
the maximum l_2-norm of the exponent vectors of all monomials. NOTE: this is not in general an integer

maximal_degree
the largest exponent in any of its monomials = the maximum l_infinity-norm of the exponent vectors of all monomials

factorisation
a tuple of factorisation_tree and factor_container. s. below

factorisation_tree
the object oriented, recursive data structure representing the factorisation (only if keep_tree=True)

factor_container
the object containing all (unique) factors of the factorisation (only if keep_tree=True)

root_value_idx
the index in the value array where the value of this polynomial (= root of the factorisation_tree) will be stored

value_array_length
the amount of addresses (storage) required to evaluate the polynomial. for evaluating the polynomial in tree form intermediary results have to be stored in a value array. the value array begins with the coefficients of the polynomial. (without further optimisation) every factor requires its own address.

copy_recipe
ndarray encoding the operations required to evaluate all scalar factors with exponent 1

scalar_recipe
ndarray encoding the operations required to evaluate all remaining scalar factors

monomial_recipe
ndarray encoding the operations required to evaluate all monomial factors

tree_recipe
ndarray encoding the addresses required to evaluate the polynomial values of the factorisation_tree.

tree_ops
ndarray encoding the type of operation required to evaluate the polynomial values of the factorisation_tree. encoded as a boolean ndarray separate from tree_recipe, since only the two operations ADD & MUL need to be encoded.

Raises

- `TypeError` – if coefficients or exponents are not given as ndarrays of the required dtype
- `ValueError` – if coefficients or exponents do not have the required shape or do not fulfill the other requirements

`__init__`(*coefficients*, *exponents*, *rectify_input: bool = False*, *compute_representation: bool = False*, *verbose: bool = False*, *keep_tree: bool = False*, *store_c_instr: bool = False*, *store_numpy_recipe: bool = False*, **args*, ***kwargs*)

root_class: Type

keep_tree: bool

value_array_length: int

recipe: Tuple

ctype_x

ctype_coeff

property factorisation_tree: BasePolynomialNode

property factor_container: FactorContainer

compute_string_representation(*coeff_fmt_str: str = '{:.2}'*, *factor_fmt_str: str = 'x_{dim}^{exp}'*, **args*, ***kwargs*) → str
computes a string representation of the polynomial and sets self.representation

Returns
a string representing this polynomial instance

eval(*x: Union[ndarray, List[float]]*, *rectify_input: bool = False*) → float
computes the value of the polynomial at query point x

either uses C or numpy+Numba evaluation

Parameters

- **x** – ndarray of floats with shape = [self.dim] representing the query point
- **rectify_input** – whether to convert coefficients and exponents into compatible numpy arrays with this set to True, the query point x can be given in standard python arrays

Returns
the value of the polynomial at point x

Raises

- `TypeError` – if x is not given as ndarray of dtype float
- `ValueError` – if x does not have the shape `[self.dim]`

`eval_complex`(*x: ndarray*) → complex128

computes the value of the polynomial at a complex query point x

Parameters
`x` – the query point given as numpy complex type

Returns
the complex value of the polynomial at point x

Raises

- `TypeError` – if x is not given as ndarray of dtype complex
- `ValueError` – if x does not have the shape `[self.dim]`

`property c_eval_fct`

`get_c_file_name`(*ending: str = '.c'*) → str

`property c_file: Path`

`property c_file_compiled`

`property recipe_file: Path`

`get_c_instructions`() → str

`factorisation`

`root_value_idx`

`use_c_eval`

`change_coefficients`(*coefficients: Union[ndarray, List[float]]*, *rectify_input: bool = False*, *compute_representation: bool = False*, *in_place: bool = False*, **args*, ***kwargs*) → AbstractPolynomial

`coefficients: ndarray`

`compute_representation: bool`

`dim: int`

`euclidean_degree: float`

`exponents: ndarray`

`export_pickle`(*path: str = 'multivar_polynomial.pickle'*)

`get_gradient`(**args*, ***kwargs*) → List[AbstractPolynomial]

Note: all arguments will be passed to the constructor of the derivative polynomials

Returns
the list of all partial derivatives

`get_partial_derivative`(*i: int*, **args*, ***kwargs*) → AbstractPolynomial

retrieves a partial derivative

Note: all given additional arguments will be passed to the constructor of the derivative polynomial

Parameters

`i` – dimension to derive with respect to. ATTENTION: dimension counting starts with 1 (i >= 1)

Returns

the partial derivative of this polynomial wrt. the i-th dimension

`maximal_degree: int`

`num_monomials: int`

`num_ops: int`

`print`(**args*)

`representation: str`

`total_degree: int`

`unused_variables`

`verbose: bool`

5.2 MultivarPolynomial

class multivar_horner.**MultivarPolynomial**(*coefficients: Union[ndarray, List[float]]*, *exponents: Union[ndarray, List[List[int]]]*, *rectify_input: bool = False*, *compute_representation: bool = False*, *verbose: bool = False*, **args*, ***kwargs*)

Bases: `AbstractPolynomial`

a representation of a multivariate polynomial in 'canonical form' (without any factorisation)

Parameters

- **coefficients** – ndarray of floats with shape (N,1) representing the coefficients of the monomials NOTE: coefficients with value 0 and 1 are allowed and will not affect the internal representation, because coefficients must be replaceable
- **exponents** – ndarray of unsigned integers with shape (N,m) representing the exponents of the monomials where m is the number of dimensions (self.dim), the ordering corresponds to the ordering of the coefficients, every exponent row has to be unique!
- **rectify_input** – bool, default=False whether to convert coefficients and exponents into compatible numpy arrays with this set to True, coefficients and exponents can be given in standard python arrays
- **compute_representation** – bool, default=False whether to compute a string representation of the polynomial
- **verbose** – bool, default=False whether to print status statements

num_monomials
the amount of coefficients/monomials N of the polynomial

dim
the dimensionality m of the polynomial NOTE: the polynomial needs not to actually depend on all m dimensions

unused_variables
the dimensions the polynomial does not depend on

num_ops
the amount of mathematical operations required to evaluate the polynomial in this representation

representation
a human readable string visualising the polynomial representation

Raises

- **TypeError** – if coefficients or exponents are not given as ndarrays of the required dtype
- **ValueError** – if coefficients or exponents do not have the required shape or do not fulfill the other requirements or `rectify_input=True` and there are negative exponents

__init__(*coefficients: Union[ndarray, List[float]]*, *exponents: Union[ndarray, List[List[int]]]*, *rectify_input: bool = False*, *compute_representation: bool = False*, *verbose: bool = False*, **args*, ***kwargs*)

num_ops: int

compute_string_representation(*coeff_fmt_str: str = '{:.2}'*, *factor_fmt_str: str = 'x_{dim}^{exp}'*, **args*, ***kwargs*) → str

computes a string representation of the polynomial and sets self.representation

Returns
a string representing this polynomial instance

eval(*x: Union[ndarray, List[float]]*, *rectify_input: bool = False*) → float

computes the value of the polynomial at query point x

makes use of fast `Numba` just in time compiled functions

Parameters

- **x** – ndarray of floats with shape = [self.dim] representing the query point
- **rectify_input** – bool, default=False whether to convert coefficients and exponents into compatible numpy arrays with this set to True, the query point x can be given in standard python arrays

Returns
the value of the polynomial at point x

Raises

- **TypeError** – if x is not given as ndarray of dtype float
- **ValueError** – if x does not have the shape `[self.dim]`

eval_complex(*x: ndarray*) → complex128

computes the value of the polynomial at a complex query point x

Parameters
x – the query point given as numpy complex type

Returns
the complex value of the polynomial at point x

Raises

- `TypeError` – if x is not given as ndarray of dtype complex
- `ValueError` – if x does not have the shape `[self.dim]`

`compute_representation: bool`

`coefficients: ndarray`

`euclidean_degree: float`

`exponents: ndarray`

`num_monomials: int`

`dim: int`

`maximal_degree: int`

`total_degree: int`

`unused_variables`

`representation: str`

`verbose: bool`

`change_coefficients`(*coefficients: Union[ndarray, List[float]]*, *rectify_input: bool = False*, *compute_representation: bool = False*, *in_place: bool = False*, **args*, ***kwargs*) → AbstractPolynomial

`export_pickle`(*path: str = 'multivar_polynomial.pickle'*)

`get_gradient`(**args*, ***kwargs*) → List[AbstractPolynomial]

Note: all arguments will be passed to the constructor of the derivative polynomials

Returns
the list of all partial derivatives

`get_partial_derivative`(*i: int*, **args*, ***kwargs*) → AbstractPolynomial
retrieves a partial derivative

Note: all given additional arguments will be passed to the constructor of the derivative polynomial

Parameters
`i` – dimension to derive with respect to. ATTENTION: dimension counting starts with 1 (i >= 1)

Returns
the partial derivative of this polynomial wrt. the i-th dimension

`print`(**args*)

5.3 HornerMultivarPolynomialOpt

class multivar_horner.**HornerMultivarPolynomialOpt**(*coefficients*, *exponents*, *rectify_input: bool = False*, *compute_representation: bool = False*, *verbose: bool = False*, *keep_tree: bool = False*, *store_c_instr: bool = False*, *store_numpy_recipe: bool = False*, **args*, ***kwargs*)

Bases: *HornerMultivarPolynomial*

a Horner factorised polynomial with an optimal factorisation found by searching all possible factorisations

Optimality in this context refers to the minimal amount of operations needed for evaluation in comparison to other Horner factorisation (not other factorisatio/optimisation techniques).

NOTES:

- **this requires MUCH more computational resources than just trying one factorisation**
  (the number of possible factorisations is growing exponentially with the size of the polynomial!).
- for the small polynomial examples in the current tests, the found factorisations were not superior

root_class: Type

factorisation

root_value_idx

value_array_length: int

keep_tree: bool

use_c_eval

recipe: Tuple

ctype_x

ctype_coeff

__init__(*coefficients*, *exponents*, *rectify_input: bool = False*, *compute_representation: bool = False*, *verbose: bool = False*, *keep_tree: bool = False*, *store_c_instr: bool = False*, *store_numpy_recipe: bool = False*, **args*, ***kwargs*)

property c_eval_fct

property c_file: Path

property c_file_compiled

change_coefficients(*coefficients: Union[ndarray, List[float]]*, *rectify_input: bool = False*, *compute_representation: bool = False*, *in_place: bool = False*, **args*, ***kwargs*) → AbstractPolynomial

coefficients: ndarray

compute_representation: bool

compute_string_representation(*coeff_fmt_str: str = '{:.2}'*, *factor_fmt_str: str = 'x_{dim}^{exp}'*, **args*, ***kwargs*) → str

computes a string representation of the polynomial and sets self.representation

Returns
a string representing this polynomial instance

dim: int

euclidean_degree: float

eval(*x: Union[ndarray, List[float]]*, *rectify_input: bool = False*) → float

computes the value of the polynomial at query point x

either uses C or numpy+Numba evaluation

Parameters

- **x** – ndarray of floats with shape = [self.dim] representing the query point
- **rectify_input** – whether to convert coefficients and exponents into compatible numpy arrays with this set to True, the query point x can be given in standard python arrays

Returns
the value of the polynomial at point x

Raises

- **TypeError** – if x is not given as ndarray of dtype float
- **ValueError** – if x does not have the shape `[self.dim]`

eval_complex(*x: ndarray*) → complex128

computes the value of the polynomial at a complex query point x

Parameters
x – the query point given as numpy complex type

Returns
the complex value of the polynomial at point x

Raises

- **TypeError** – if x is not given as ndarray of dtype complex
- **ValueError** – if x does not have the shape `[self.dim]`

exponents: ndarray

export_pickle(*path: str = 'multivar_polynomial.pickle'*)

property factor_container: FactorContainer

property factorisation_tree: BasePolynomialNode

get_c_file_name(*ending: str = '.c'*) → str

get_c_instructions() → str

get_gradient(**args*, ***kwargs*) → List[AbstractPolynomial]

Note: all arguments will be passed to the constructor of the derivative polynomials

Returns
the list of all partial derivatives

`get_partial_derivative`(*i: int*, **args*, ***kwargs*) → AbstractPolynomial

retrieves a partial derivative

Note: all given additional arguments will be passed to the constructor of the derivative polynomial

Parameters
`i` – dimension to derive with respect to. ATTENTION: dimension counting starts with 1 (i >= 1)

Returns
the partial derivative of this polynomial wrt. the i-th dimension

`maximal_degree: int`

`num_monomials: int`

`num_ops: int`

`print`(**args*)

`property recipe_file: Path`

`representation: str`

`total_degree: int`

`unused_variables`

`verbose: bool`

CHAPTER

SIX

CONTRIBUTION GUIDELINES

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs via Github Issues.

If you are reporting a bug, please include:

- Your version of this package, python and Numba (if you use it)
- Any other details about your local setup that might be helpful in troubleshooting, e.g. operating system.
- Detailed steps to reproduce the bug.
- Detailed description of the bug (error log etc.).

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "help wanted" and not assigned to anyone is open to whoever wants to implement it - please leave a comment to say you have started working on it, and open a pull request as soon as you have something working, so that Travis starts building it.

Issues without "help wanted" generally already have some code ready in the background (maybe it's not yet open source), but you can still contribute to them by saying how you'd find the fix useful, linking to known prior art, or other such help.

6.1.4 Write Documentation

Probably for some features the documentation is missing or unclear. You can help with that!

6.1.5 Submit Feedback

The best way to send feedback is to file an issue via Github Issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement. Create multiple issues if necessary.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up this package for local development.

- Fork this repo on GitHub.
- Clone your fork locally
- To make changes, create a branch for local development:

  ```
  $ git checkout -b name-of-your-bugfix-or-feature
  ```

- Check out the instructions and notes in `publish.py`
- Install `tox` and run the tests:

  ```
  $ pip install tox
  $ tox
  ```

  The `tox.ini` file defines a large number of test environments, for different Python etc., plus for checking codestyle. During development of a feature/fix, you'll probably want to run just one plus the relevant codestyle:

  ```
  $ tox -e codestyle
  ```

- Commit your changes and push your branch to GitHub:

  ```
  $ git add .
  $ git commit -m "Your detailed description of your changes."
  $ git push origin name-of-your-bugfix-or-feature
  ```

- Submit a pull request through the GitHub website. This will trigger the Travis CI build which runs the tests against all supported versions of Python.

CHAPTER

SEVEN

CHANGELOG

TODOs

- build html docs and include with package: "docs/_build/html/*"
- run speed and numerical tests with the new C evaluation method!
- Improve tests
- compare poly.num_ops of different factorisations. tests?
- num_ops currently will be 0 when caching is used (no factorisation will be computed)

POSSIBLE IMPROVEMENTS:

MultivarPoly (unfactorised):

- **also make use of the concept of 'recipes' for efficiently evaluating the polynomial**
  skipping the most unnecessary operations (actually more fair comparison in terms of operations required for evaluation)
- add option to skip this optimisation

HornerMultivarPoly:

- **optimise factor evaluation (save instructions, 'factor factorisation'):**
  a monomial factor consists of scalar factors and in turn some monomial factors consist of other monomial factors

-> the result of evaluating a factor can be reused for evaluating other factors containing it

-> find the optimal 'factorisation' of the factors themselves

-> set the factorisation_idxs of each factor in total_degree to link the evaluation appropriately

idea:
  choose 'Goedel IDs' as the monomial factor ids then the id of a monomial is the product of the ids of its scalar factors find the highest possible divisor among all factor ids (corresponds to the 'largest' factor included in the monomial) this leads to a minimal factorisation for evaluating the monomial values quickly

- add option to skip this optimisation to save build time
- **optimise space requirement:**
  after building a factorisation tree for the factors themselves, then use its structure to cleverly reuse storage space -> use compiler construction theory: minimal assembler register assignment, 'graph coloring'…
- **optimise 'copy recipe': avoid copy operations for accessing values of x**

  problem: inserting x into the value array causes operations as well and
    complicates address assigment and recipe compilation

- **when the polynomial does not depend on all variables, build a wrapper to maintain the same "interface"**
  but internally reduce the dimensionality, this reduced the size of the numpy arrays -> speed, storage benefit
- **the evaluation of subtrees is independent and could theoretically be done in parallel**
  probably not worth the effort. more reasonable to just evaluate multiple polynomials in parallel

7.1 3.1.0 (2023-02-15)

- supporting evaluation at complex query points as requested in issue #37

internal: * hypothesis tests: evaluation equality of Horner and regular polynomials (up to numerical errors) * added mypy typing pre-commit hook

7.2 3.0.5 (2022-12-10)

- bump pytest dependency version to `>=7,<8` (vulnerability fix)
- less strict dependency pinning to support python 3.11
- added python 3.11 tests (not yet supporting numba)
- disabled numerical stability tests due to missing numpy support for high precision 128-bit float on arm64 architecture

7.3 3.0.4 (2022-07-10)

- bump numpy dependency version to `1.22` (vulnerability fix)
- officially supported python versions `>=3.8,<3.11` (due to numpy and numba constraints)

7.4 3.0.3 (2022-06-15)

- bugfix: packaging. now completely based on pyproject.toml (poetry)

7.5 3.0.2 (2022-06-14)

- bugfix: optional `numba` dependency. numba imports were not optional
- bugfix: create __cache__ dir if not exists
- minor documentation improvements
- bumping dependencies

7.6 3.0.1 (2021-12-04)

ATTENTION: major changes:

- introduced the default behavior of compiling the evaluation instructions in C code (C compiler required)
- the previous `numpy+numba` evaluation using "recipes" is the fallback option in case the C file could not be compiled
- as a consequence dropping `numba` as a required dependency
- added the "extra" `numba` to install on demand with: `pip install multivar_horner[numba]`
- introduced custom polynomial hashing and comparison operations
- using hash to cache and reuse the instructions for evaluation (for both C and recipe instructions)
- introduced constructions argument `store_c_instr` (`HornerMultivarPolynomial`) to force the storage of evaluation code in C for later usage
- introduced constructions argument `store_numpy_recipe` (`HornerMultivarPolynomial`) to force the storage of the custom "recipe" data structure required for the evaluation using `numpy` and `numba`
- introduced class `HornerMultivarPolynomialOpt` for optimal Horner Factorisations to separate code and simplify tests
- as a consequence dropped construction argument `find_optimal` of class `HornerMultivarPolynomial`
- introduced constructions argument `verbose` to show the output of status print statements
- dropping official python3.6 support because `numba` did so (supporting Python3.7+)

internal:

- using poetry for dependency management
- using GitHub Actions for CI instead of travis

7.7 2.2.0 (2021-02-04)

ATTENTION: API changes:

- removed `validate_input` arguments. input will now always be validated (otherwise the numba jit compiled functions will fail with cryptic error messages)
- black code style
- pre-commit checks

7.8 2.1.1 (2020-10-01)

Post-JOSS paper review release:

- Changed the method of counting the amount of operations of the polynomial representations. Only the multiplications are being counted. Exponentiations count as (exponent-1) operations.
- the numerical tests compute the relative average error with an increased precision now

7.9 2.1.0 (2020-06-15)

ATTENTION: API changes:

- `TypeError` and `ValueError` are being raised instead of `AssertionError` in case of invalid input parameters with `validate_input=True`
- added same parameters and behavior of `rectify_input` and `validate_input` in the `.eval()` function of polynomials

internal:

- Use `np.asarray()` instead of `np.array()` to avoid unnecessary copies
- more test cases for invalid input parameters

7.10 2.0.0 (2020-04-28)

- BUGFIX: factor evaluation optimisation caused errors in rare cases. this optimisation has been removed completely. every factor occurring in a factorisation is being evaluated independently now. this simplifies the factorisation process. the concept of "Goedel ID" (=unique encoding using prime numbers) is not required any more
- ATTENTION: changed polynomial degree class attribute names to comply with naming conventions of the scientific literature
- added __call__ method for evaluating a polynomial in a simplified notation `v=p(x)`
- fixed dependencies to: `numpy>=1.16`, `numba>=0.48`
- clarified docstrings (using Google style)
- more verbose error messages during input verification
- split up `requirements.txt` (into basic dependencies and test dependencies)
- added sphinx documentation
- updated benchmark results

tests:

- added test for numerical stability
- added plotting features for evaluating the numerical stability
- added tests comparing functionality to 1D `numpy` polynomials
- added tests comparing functionality to naive polynomial evaluation
- added basic API functionality test

internal:

- added class `AbstractPolynomial`
- added typing
- adjusted publishing routine
- testing multiple python versions
- using the specific tags of the supported python version for the build wheels
- removed `example.py`

7.11 1.3.0 (2020-03-14)

- NEW FEATURE: changing coefficients on the fly with `poly.change_coefficients(coeffs)`
- NEW DEPENDENCY: `python3.6+` (for using f" format strings)
- the real valued coefficients are now included in the string representation of a factorised polynomial
- add contribution guidelines
- added instructions in readme, `example.py`
- restructured the factorisation routine (simplified, clean up)
- extended tests

7.12 1.2.0 (2019-05-19)

- support of newer numpy versions (ndarray.max() not supported)
- added plotting routine (partly taken from tests)
- added plots in readme
- included latest insights into readme

7.13 1.1.0 (2019-02-27)

- added option *find_optimal* to find an optimal factorisation with A* search, explanation in readme
- optimized heuristic factorisation (more clean approach using just binary trees)
- dropped option *univariate_factors*
- added option *compute_representation* to compute the string representation of a factorisation only when required
- added option *keep_tree* to keep the factorisation tree when required
- clarification and expansion of readme and *example.py*
- explained usage of optional parameters *rectify_input=True* and *validate_input=True*
- explained usage of functions *get_gradient()* and *get_partial_derivative(i)*
- averaged runtime in speed tests

7.14 1.0.1 (2018-11-12)

- introducing option to only factor out single variables with the highest usage with the optional parameter `univariate_factors=True`
- compute the number of operations needed by the horner factorisation by the length of its recipe (instead of traversing the full tree)
- instead of computing the value of scalar factors with exponent 1, just copy the values from the given x vector ("copy recipe")
- compile the initial value array at construction time

7.15 1.0.0 (2018-11-08)

- first stable release

7.16 0.0.1 (2018-10-05)

- birth of this package

CHAPTER

EIGHT

REFERENCES

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] William George Horner. Xxi. a new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, pages 308–335, 1819.

[2] Martine Ceberio and Vladik Kreinovich. Greedy algorithms for optimizing multivariate horner schemes. *ACM SIGSAM Bulletin*, 38(1):8–15, 2004.

[3] Juan Manuel Peña and Thomas Sauer. On the multivariate Horner scheme. *SIAM journal on numerical analysis*, 37(4):1186–1197, 2000.

[4] Juan Manuel Peña and Thomas Sauer. On the multivariate Horner scheme II: running error analysis. *Computing*, 65(4):313–322, 2000.

[5] Lloyd Trefethen. Multivariate polynomial approximation in the hypercube. *Proceedings of the American Mathematical Society*, 145(11):4837–4844, 2017.

[6] Michael Hecht, Karl B Hoffmann, Bevan L Cheeseman, and Ivo F Sbalzarini. Multivariate Newton interpolation. *arXiv preprint arXiv:1812.04256*, 2018.

[7] Michael Hecht and Ivo F. Sbalzarini. Fast interpolation and Fourier transform in high-dimensional spaces. In K. Arai, S. Kapoor, and R. Bhatia, editors, *Intelligent Computing. Proc. 2018 IEEE Computing Conf., Vol. 2,*, volume 857 of Advances in Intelligent Systems and Computing, 53–75. London, UK, 2018. Springer Nature.

[8] J Carnicer and M Gasca. Evaluation of multivariate polynomials and their derivatives. *Mathematics of Computation*, 54(189):231–243, 1990.

[9] Jan Kuipers, Aske Plaat, JAM Vermaseren, and H Jaap van den Herik. Improving multivariate Horner schemes with Monte Carlo tree search. *Computer Physics Communications*, 184(11):2391–2395, 2013.

[10] Martin Mok-Don Lee. *Factorization of multivariate polynomials*. PhD thesis, Technische Universität Kaiserslautern, 2013.

[11] Charles E Leiserson, Liyun Li, Marc Moreno Maza, and Yuzhen Xie. Efficient evaluation of large polynomials. In *International Congress on Mathematical Software*, 342–353. Springer, 2010.

[12] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[13] Masakazu Kojima. Efficient evaluation of polynomials and their partial derivatives in homotopy continuation methods. *Journal of the Operations Research Society of Japan*, 51(1):29–54, 2008.

PYTHON MODULE INDEX

m

INDEX

Symbols

C

D

E

F

G

H

K

M

N

P

R

S

T

U

V