Power Series Arithmetic with the BPAS Library

Basic Polynomial Algebra Subprograms

Alexander Brandt, Mahsa Kazemi, Marc Moreno Maza

Ontario Research Center for Computer Algebra
Department of Computer Science
University of Western Ontario, Canada

CASC 2020, September 14, 2020

Past, Present, Future

The **Basic Polynomial Algebra Subprograms (BPAS)** library [3] provides support for high-performance polynomial algebra.

→ At CASC 2018 we presented sparse polynomial arithmetic [4, 6]

→ These polynomials were employed in a polynomial system solving framework based on regular chains [5]

In this talk we present our high-performance implementation of **multivariate power series** written in C.

We are motivated by: (see [1])

→ Computation of limits of multivariate rational functions

→ New applications of Hensel lifting: Extended Hensel Construction, Jung-Abhyankar Theorem

→ Computation of topological closures, resolution of singularities

Outline

Goals and Previous Work

Our goal is a high-performance power series implementation

- → a lazy implementation in a compiled language (for performance)
- → ability to exploit opportunities for concurrent programming

Lazy evaluation is not new:

- → univariate power series in Scratchpad II using Lisp [7]
- → univariate power series and relaxed algorithms [8]
- → polynomial arithmetic [10]

Yet, no general implementation of (compiled) multivariate power series

- → SageMath provides **truncated** multivariate power series
- → multivariate power series in `PowerSeries` library of Maple [2, 9]

What is a power series?

Let $\mathbf{k}$ be a field (often algebraic closed) then $\mathbf{k}[[X_1, \ldots, X_n]]$ is the **ring of formal power series**

→ indeterminates are $X_1, \ldots, X_n$, coefficients in $\mathbf{k}$

Let $f = \sum_e a_e X^e \in \mathbf{k}[[X_1, \ldots, X_n]]$

→ $a_e \in \mathbf{k}$

→ $e = (e_1, \ldots, e_n)$ is a multi-index with n coordinates

→ $|e| = e_1 + \ldots + e_n$

→ **homogeneous part**: $f_{(d)} = \sum_{|e|=d} a_e X^e$

→ **polynomial part**: $f^{(d)} = \sum_{k \le d} f_{(k)}$

Example: $f = 1 + X_1 + X_1 X_2 + X_2^2 + X_1 X_2^2 + X_1^3 + \cdots$

$$f_{(2)} = X_1 X_2 + X_2^2 \qquad\qquad f^{(2)} = 1 + X_1 + X_1 X_2 + X_2^2$$

We say f is *known to precision 3*

Outline

Design Motivations

1. Only compute terms explicitly needed: requested by user, needed for subsequent operations

2. Ability to **resume** and increase precision of an existing power series

This suggests the need for:

→ **power series ancestry**, a history of operands and operators leading to a particular "child" power series

→ **generator functions**, a function to produce new terms of a power series on demand

Ancestry Example

$$
\begin{tikzcd}[column sep=small]
\begin{array}{l} f = \\ 1 + x + yz + \ldots \end{array} \arrow[dr] & & \begin{array}{l} g = \\ 1 + z + y + \ldots \end{array} \arrow[dl] & & \begin{array}{l} a = \\ 1 + y + x^2 + \ldots \end{array} \arrow[dr] & & \begin{array}{l} b = \\ 1 + yz + xz + \ldots \end{array} \arrow[dl] \\
& \times \arrow[d] & & & & \times \arrow[d] & \\
& \begin{array}{l} h = \\ 1 + z + y + x + yz + xz + xy + \ldots \end{array} \arrow[drr] & & & & \begin{array}{l} c = \\ 1 + y + yz + xz + x^2 + \ldots \end{array} \arrow[dll] & \\
& & & + \arrow[d] & & & \\
& & & \begin{array}{l} h + c = \\ 2 + z + 2y + x + 2yz + 2xz + xy + x^2 + \ldots \end{array} & & &
\end{tikzcd}
$$

Generator Functions

Generators could be **co-routines** or **iterators**, continually **yield**-ing terms of a power series in increasing order.

→ Power series operations/arithmetic also necessitates dynamic combinations of generator functions

→ Easy in a scripting language, Harder in a compiled language

In a more "closed-form" solution, our generators:

→ **generate a homogeneous part** of a power series, for a particular (total) degree, where

→ the **degree is a parameter** of the function

Top-level `homogeneous_part` and `polynomial_part` functions call the generators generically, as needed for particular degrees

Encoding a Power Series

Our power series struct:

- → **dense array** of homogeneous polynomials

$f_{(0)}$	$f_{(1)}$	$f_{(2)}$	$f_{(3)}$	$f_{(4)}$	$f_{(5)}$	$\cdots$

- → `int`'s for current allocation, precision
- → a function pointer to a **generator**
- → the arguments to pass to the generator function.

The struct emulates a **function closure** for the generator:

- → captures and stores all necessary variables by reference (pointer) to pass as arguments to the generator
- → uses `void*` parameters for generality
- → The ancestry is implied by storing power series pointers as parameters
  - ↳ use **reference counting** on the power series

The PowerSeries struct

```c
typedef Poly_ptr (*homog_part_gen)(int);
typedef Poly_ptr (*homog_part_gen_unary)(int, void*);
typedef Poly_ptr (*homog_part_gen_binary)(int, void*, void*);
typedef Poly_ptr (*homog_part_gen_tert)(int,void*,void*,void*);

typedef union HomogPartGenerator {
    homog_part_gen nullaryGen;
    homog_part_gen_unary unaryGen;
    homog_part_gen_binary binaryGen;
    homog_part_gen_tert tertiaryGen;
} HomogPartGenerator_u;

typedef struct PowerSeries {
    int deg, alloc;
    Poly_ptr* homog_polys;

    HomogPartGenerator_u gen;
    int genOrder;
    void *genParam1, *genParam2, *genParam3;

    int refCount;
} PowerSeries_t;
```

Power Series Arithmetic: Multiplication

1. A top-level lazy function sets up a `PowerSeries` with generator and generator parameters and immediately returns
2. A **void generator** (wrapper function) is called generically with the `void*` params, casting them to the correct type, and then calls...
3. The **true generator** creates and returns $f_{(d)}$ for input d :

```
Poly_ptr homogPart_prod(int d, PowerSeries_t* f, PowerSeries_t* g){
    Poly_ptr sum = zeroPolynomial();
    for (int i = 0; i <= d; i++) {
        Poly_ptr p = multPolys(homogPart(d-i,f), homogPart(i,g));
        sum = addPolynomials(sum, p);
    }
    return sum;
}
```

→ "Top-level" `homogPart` immediately returns already computed terms, or calls the generator through the function pointer as needed

→ Other supported operations: addition, subtraction, negation, inversion

The Ancestry and Generators

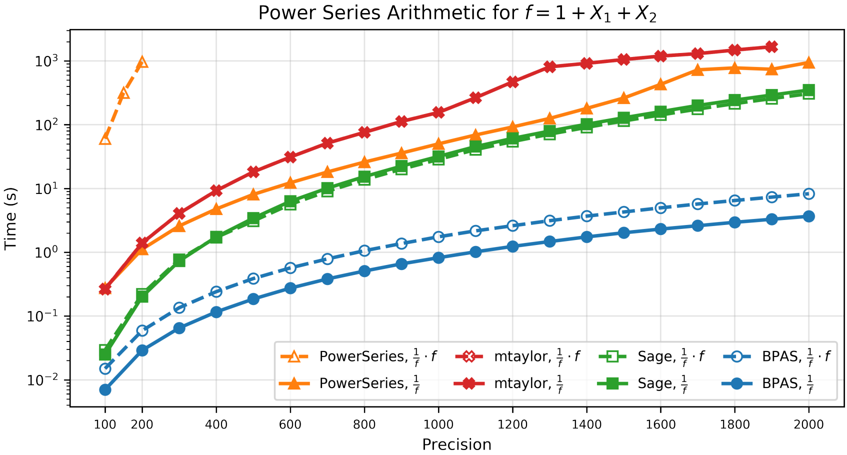
The power series ancestry is implied by a **generator's parameters**

- → For a power series `f`, `f.genParam1`, `f.genParam2`, ... are its **parents**
- → Relationship is one-sided; parents don't know about their children

For a generator to make use of its parents, they must be kept "alive"

- → **reference counting**
- → a parent's reference count is incremented when a child is created
- → "destroying" only decrements reference count...
- → when count ≤ 0, then data is actually `free`'d
- → when a child is `free`'d, its parents get "destroyed"

Experimentation: Integer coefs, 2 vars

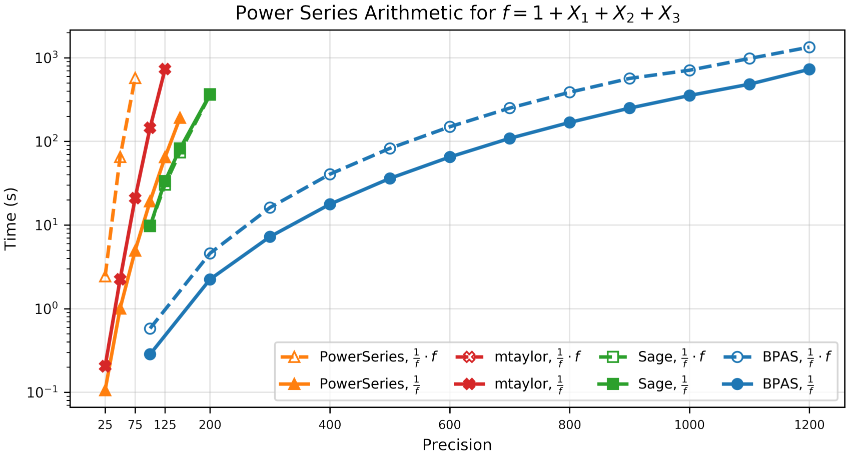


→ `PowerSeries` library in MAPLE [2, 9]

→ `mtaylor` in MAPLE 2020

→ Truncated multivariate power series in SAGE

Experimentation: Integer coefs, 3 vars

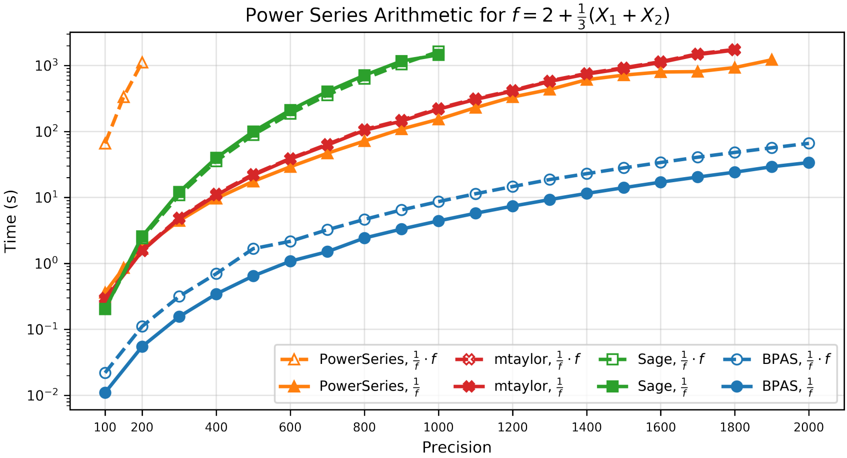


→ `PowerSeries` library in MAPLE [2, 9]

→ `mtaylor` in MAPLE 2020

→ Truncated multivariate power series in SAGE

Experimentation: Rat. Num coefs, 2 vars



- → `PowerSeries` library in MAPLE [2, 9]
- → `mtaylor` in MAPLE 2020
- → Truncated multivariate power series in SAGE

Outline

Weierstrass Preparation Theorem

Let $\mathbb{A} = \mathbf{k}[[X_1, \ldots, X_n]]$ and $\mathcal{M} = \langle X_1, \ldots, X_n \rangle$ be its maximal ideal

Let $f = \sum_i a_i Y^i \in \mathbb{A}[[Y]]$, $d \geq 0$ be the smallest integer s.t. $a_d \notin \mathcal{M}$

→ at the origin $(X_1 = \cdots = X_n = 0)$, $f \neq 0$ and $a_d \neq 0$

WPT yields a **polynomial approximation of a power series** around the origin through factorization

Weierstrass Preparation Theorem: there exists unique $\alpha,\ p$ s.t.

(i) $\alpha \in \mathbb{A}[[Y]]$ is invertible,

(ii) $p = Y^d + \sum_{i=0}^{d-1} b_i Y^i \in \mathbb{A}[Y]$ with $b_0, \ldots, b_{d-1} \in \mathcal{M}$

(iii) $f = \alpha p$

↳ $p \in \mathbb{A}[Y]$ is a monic Univariate Polynomial over Power Series, *UPoPS*

↳ if f is a UPoPS then so is α

Computability of WPT: A Lemma

Let $f, g, h \in \mathbb{A}$ with $f = gh$. With f and h known, compute g.

Assume $f_{(0)} = 0$ and $h_{(0)} \neq 0$, then:

$$f_{(0)} = g_{(0)} h_{(0)} = 0 \implies g_{(0)} = 0$$

By induction, $g_{(r)}$ is uniquely determined by $f_{(1)}, \ldots, f_{(r)}, h_{(0)}, \ldots, h_{(r-1)}$

$$\begin{aligned}
f_{(1)} + f_{(2)} + \cdots + f_{(r)} &= (g_{(1)} + g_{(2)} + \cdots + g_{(r)})(h_{(0)} + h_{(1)} + \cdots + h_{(r)}) \\
f_{(1)} &= g_{(1)} h_{(0)} \\
f_{(2)} &= g_{(2)} h_{(0)} + g_{(1)} h_{(1)} \\
&\vdots \\
f_{(r)} &= g_{(r)} h_{(0)} + g_{(r-1)} h_{(1)} + \cdots + g_{(1)} h_{(r-1)}
\end{aligned}$$

We can compute $g_{(r)}$ for $r = 1, 2, \ldots$ **using only polynomial arithmetic**:

$$\frac{1}{h_{(0)}} \left(f_{(r)} - g_{(r-1)} h_{(1)} - \cdots - g_{(1)} h_{(r-1)} \right) = g_{(r)}$$

Lazy Weierstrass Preparation

Let $f = \sum_{\ell}^{d+m} a_\ell Y^\ell$, $p = Y^d + \sum_j^{d-1} b_j Y^j$, $\alpha = \sum_i^m c_i Y^i$ be UPoPS.

↳ a_ℓ, b_j, c_i are power series

↳ $b_j \in \mathcal{M}$ for $j = 0, \ldots, d-1$

$$f = \alpha p \implies \begin{array}{rcl} a_0 & = & b_0 c_0 \\ a_1 & = & b_0 c_1 + b_1 c_0 \\ & \vdots & \\ a_{d-1} & = & b_0 c_{d-1} + b_1 c_{d-2} + \cdots + b_{d-2} c_1 + b_{d-1} c_0 \\ a_d & = & b_0 c_d + b_1 c_{d-1} + \cdots + b_{d-1} c_1 + c_0 \\ & \vdots & \\ a_{d+m-1} & = & b_{d-1} c_m + c_{m-1} \\ a_{d+m} & = & c_m \end{array}$$

We update p and α by solving these equations modulo $\mathcal{M}^r$, $r = 1, 2, \ldots$

↳ "ping-pong" updates: p to mod $\mathcal{M}^2$, α to mod $\mathcal{M}^2$, p to mod $\mathcal{M}^3 \ldots$

(1) $b_j \equiv 0 \mod \mathcal{M}$, $j = 0, \ldots, d-1$ (2) $c_i \equiv a_i \mod \mathcal{M}$ for $i = 0, \ldots, m$

Lazy Weierstrass Phase 1: Updating p

Let $f = \sum_{\ell}^{d+m} a_\ell Y^\ell$, $p = Y^d + \sum_j^{d-1} b_j Y^j$, $\alpha = \sum_i^m c_i Y^i$ be UPoPS.

↳ a_ℓ, b_j, c_i are power series ↳ $b_j \in \mathcal{M}$ for $j = 0, \ldots, d-1$

$$
\begin{aligned}
a_0 &= b_0 c_0 \\
a_1 - b_0 c_1 &= b_1 c_0 \\
a_2 - b_0 c_2 - b_1 c_1 &= b_2 c_0 \\
&\vdots \\
a_{d-1} - b_0 c_{d-1} - b_1 c_{d-2} + \cdots - b_{d-2} c_1 &= b_{d-1} c_0
\end{aligned}
$$

$b_j \equiv 0 \mod \mathcal{M}$, $j = 0, \ldots, d-1$. Then, for $\mathcal{M}^r$, $r > 1$:

- → let $F_j = a_j - \sum_{k=0}^{j-1} b_k c_{i-k}$
- → the previous lemma applies to each equation $F_j = b_j c_0$ to update each b_j **in succession**, from $j = 0$ to $d-1$
- → Each F_j automatically updated through updated b_k and lazy power series arithmetic

Lazy Weierstrass Phase 2: Updating α

Let $f = \sum_{\ell}^{d+m} a_\ell Y^\ell$, $p = Y^d + \sum_{j}^{d-1} b_j Y^j$, $\alpha = \sum_{i}^{m} c_i Y^i$ be UPoPS.

↳ a_ℓ, b_j, c_i are power series

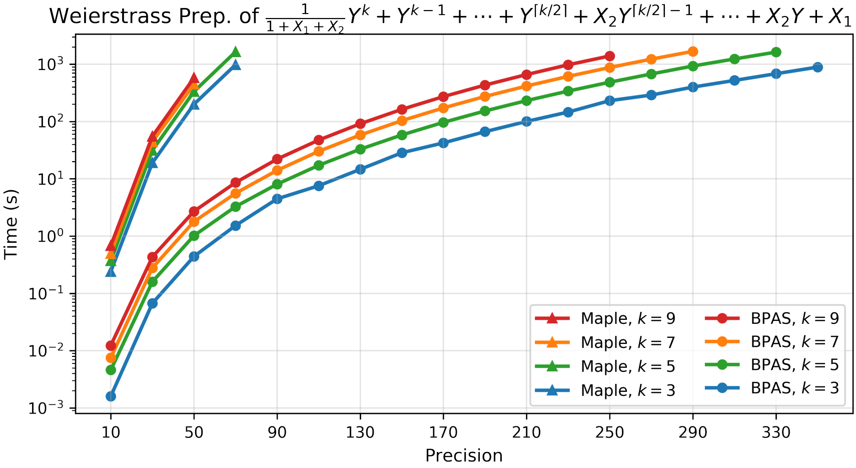
↳ $b_j \in \mathcal{M}$ for $j = 0, \ldots, d-1$

$$\begin{array}{rcl} c_m & = & a_{d+m} \\ c_{m-1} & = & a_{d+m-1} - b_{d-1}c_m \\ c_{m-2} & = & a_{d+m-2} - b_{d-2}c_m - b_{d-1}c_{m-1} \\ & \vdots & \\ c_0 & = & a_d - b_0 c_d - b_1 c_{d-1} - \cdots - b_{d-1}c_1 \end{array}$$

$c_i \equiv a_i \mod \mathcal{M}$ for $i = 0, \ldots, m$. Then, for $\mathcal{M}^r$, $r > 1$:

- → In Phase 1, b_j, $j = 0, \ldots, d-1$ updated to modulo $\mathcal{M}^r$
- → c_i then automatically updated (by lazy arithmetic) to modulo $\mathcal{M}^r$
- → Note: updating each c_i is independent since $b_j \in \mathcal{M}$.
  e.g. b_j known $\mathcal{M}^r$, c_i known $\mathcal{M}^{r-1} \implies b_j c_i$ known modulo $\mathcal{M}^r$

Experimentation



Outline

Hensel's Lemma

Let $f = \sum_i^k a_i Y^i \in \mathbf{k}[[X_1, \ldots, X_n]][Y]$ be a monic polynomial of degree k.

Let $\bar{f} = f(0, \ldots, 0, Y) \in \mathbf{k}[Y]$. Assuming $\mathbf{k}$ is algebraically closed, $\bar{f}$ factorizes into linear factors $\bar{f} = (Y - c_1)^{k_1} \cdots (Y - c_r)^{k_r}$.

Hensel's Lemma: there exists monic $f_1, \ldots, f_r \in \mathbf{k}[[X_1, \ldots, X_n]][Y]$ s.t.

(i) $f = f_1 \cdots f_r$,

(ii) $\deg(f_i, Y) = k_i$, for $i = 0, \ldots, r$

(iii) $\bar{f}_i = (Y - c_i)^{k_i}$, for $i = 0, \ldots, r$

A factorization routine:

1. Translate f by c_i, it now has order k_i
2. Weierstrass preparation can then be applied to obtain p with degree k_i and α with degree $k - k_i$
3. After the reverse translation, p is f_i, and α is the "new" f

Factorization via Hensel's Lemma

Algorithm 1 HenselFactorization(f)

Input: $f = \sum_{i=0}^{k} a_i Y^i, a_i \in \mathbf{k}[[X_1, \ldots, X_n]]$.
Output: $f_1, \ldots, f_r$ satisfying Hensel's Lemma

1: $\bar{f} = f(0, \ldots, 0, Y)$
2: $c_1, \ldots, c_r \leftarrow$ obtain roots of $\bar{f}$ in $\mathbf{k}$ ▷ factor $\bar{f}$
3: $f^* = f$
4: **for** $i = 1$ to r **do**
5: $\quad g \leftarrow f^*(Y + c_i)$
6: $\quad p, \alpha \leftarrow$ WeierstrassPreparation(g)
7: $\quad f_i \leftarrow p(Y - c_i)$
8: $\quad f^* \leftarrow \alpha(Y - c_i)$
9: **return** $f_1, \ldots, f_r$

→ The Taylor shifts $f^*(Y + c_i)$, $p(Y - c_i)$, $\alpha(Y - c_i)$ are implemented lazily. Combined with lazy WPT, this entire factorization is lazy.

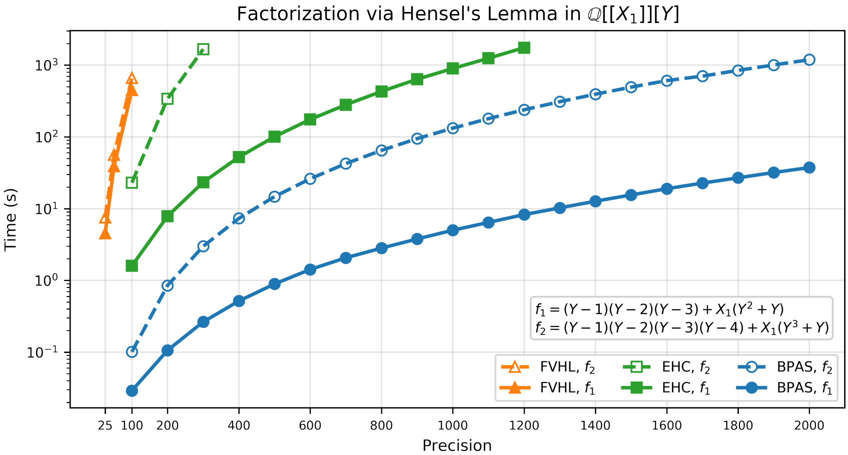
A Factorization Pipeline (w.i.p.)

Viewing factorization via Hensel's lemma as a pipelined computation provides opportunities for parallelism.

- → Updating f_i automatically updates its corresponding α, thus allowing f_{i+1} to be updated
- → Perform each Weierstrass update (and reverse shift) as stages in a parallel pipeline

	Stage 1 (f_1)	Stage 2 (f_2)	Stage 3 (f_3)	Stage 4 (f_4)
Time 1	f_1 to prec. 1			
Time 2	f_1 to prec. 2	f_2 to prec. 1		
Time 3	f_1 to prec. 3	f_2 to prec. 2	f_3 to prec. 1	
Time 4	f_1 to prec. 4	f_2 to prec. 3	f_3 to prec. 2	f_4 to prec. 1
Time 5	f_1 to prec. 5	f_2 to prec. 4	f_3 to prec. 3	f_4 to prec. 2
Time 6	f_1 to prec. 6	f_2 to prec. 5	f_3 to prec. 4	f_4 to prec. 3

Experimentation



From `PowerSeries` library:

→ FVHL: Factorization via Hensel's Lemma

→ EHC: Extended Hensel Construction (Puiseux series)

Conclusions and Future Work

We have implemented high-performing lazy implementations of Power Series and UPoPS, including:

- → Power series arithmetic: $\pm$, $\times$, $\div$
- → Weierstrass preparation
- → Taylor shift by elements of $\mathbf{k}$
- → Factorization via Hensel's lemma

Further performance to be obtained through:

- → Parallelization internal to a Weierstrass Update
- → Pipelined computation in factorization via Hensel's lemma
- → Relaxed algorithms [8]

Thank You!

Questions?

References

[1] P. Alvandi, M. Ataei, M. Kazemi, and M. Moreno Maza. "On the Extended Hensel Construction and its application to the computation of real limit points". In: *J. Symb. Comput.* 98 (2020), pp. 120–162.

[2] P. Alvandi, M. Kazemi, and M. Moreno Maza. "Computing limits with the regularchains and powerseries libraries: from rational functions to Zariski closure". In: *ACM Commun. Comput. Algebra* 50.3 (2016), pp. 93–96.

[3] M. Asadi, A. Brandt, C. Chen, S. Covanov, M. Kazemi, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, D. Talaashrafi, L. Wang, N. Xie, and Y. Xie. *Basic Polynomial Algebra Subprograms (BPAS)*. `www.bpaslib.org`. 2020.

[4] M. Asadi, A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Sparse Polynomial Arithmetic with the BPAS Library". In: *Computer Algebra in Scientific Computing, CASC 2018, Lille, France, Proceedings*. 2018, pp. 32–50.

[5] M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Y. Xie. "On the Parallelization of Triangular Decomposition of Polynomial Systems". In: *International Symposium on Symbolic and Algebraic Computation, ISSAC 2020, Proceedings*. ACM, 2020, pp. 22–29.

[6] M. Asadi, A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Algorithms and Data Structures for Sparse Polynomial Arithmetic". In: *Mathematics* 7.5 (2019), p. 441.

[7] W. H. Burge and S. M. Watt. "Infinite structures in Scratchpad II". In: *European Conference on Computer Algebra*. Springer. 1987, pp. 138–148.

[8] J. van der Hoeven. "Relax, but Don't be Too Lazy". In: *J. Symb. Comput.* 34.6 (2002), pp. 479–542.

[9] M. Kazemi and M. Moreno Maza. "Detecting Singularities Using the PowerSeries Library". In: *Maple in Mathematics Education and Research - Third Maple Conference, MC 2019, Proceedings*. Springer, 2019, pp. 145–155.

[10] M. B. Monagan and P. Vrbik. "Lazy and Forgetful Polynomial Arithmetic and Applications". In: *Computer Algebra in Scientific Computing, 11th International Workshop, CASC 2009, Proceedings*. 2009, pp. 226–239.