Power Series Arithmetic with the BPAS Library

Alexander Brandt[1] Mahsa Kazemi[2] Marc Moreno Maza[3]

Department of Computer Science, The University of Western Ontario, London, Canada
[1] abrandt5@uwo.ca, [2] mkazemin@uwo.ca, [3] moreno@csd.uwo.ca

Abstract

We discuss the design and implementation of multivariate power series, univariate polynomials over power series, and their associated arithmetic operations within the Basic Polynomial Algebra Subprograms (BPAS) Library. This implementation employs lazy variations of Weierstrass preparation and the factorization of univariate polynomials over power series following Hensel's lemma. Our implementation is lazy in that power series terms are only computed when explicitly requested. The precision of a power series is dynamically extended upon request, without requiring any re-computation of existing terms. This design extends into an "ancestry" of power series whereby power series created from the result of arithmetic or Weierstrass preparation automatically hold on to enough information to dynamically update themselves to higher precision using information from their "parents".

Keywords: lazy power series · Weierstrass preparation · Hensel's lemma

1 Introduction

Power series are polynomial-like objects with, potentially, an infinite number of terms. They play a fundamental role in theoretical computer science, functional analysis, computer algebra, and algebraic geometry. Of course, the fact that power series may have an infinite number of terms presents interesting challenges to computer scientists. How to represent them on a computer? How to perform arithmetic operations effectively and efficiently with them?

One standard approach is to implement power series as *truncated power series*, that is, by setting up in advance a sufficiently large accuracy, or precision, and discarding any power series term with a degree equal or higher to that accuracy. Unfortunately, for some important applications, not only is such accuracy problem-specific, but sometimes cannot be determined before calculations start, or later may be found to not go far enough. This scenario occurs, for instance, with modular methods [13] for polynomial system solving [7] based on Hensel lifting and its variants [22]. It is necessary then to implement power series with data structures and techniques that allow for reactivity and dynamic updates.

Since a power series has potentially infinitely many terms, it is natural to represent it as a function, that we shall call a *generator*, which computes the terms of that power series for a given accuracy. This point of view leads to natural algorithms for performing arithmetic operations (addition, multiplication, division) on power series based on *lazy evaluation*.

Another advantage of this functional approach is the fact that it supports concurrency in a natural manner. Consider a procedure which takes some number of power series as input and returns a number of power series. Assume the generators of the outputs can be determined in essentially constant time, which is often the case. Subsequent computations involving those output power series can then start almost immediately. In other words, the first procedure call is essentially non-blocking, and the output power series can (*i*) be used immediately as input to other procedure calls, and (*ii*) have their terms computed only as needed. This approach allows for power series terms to be computed or "produced" while concurrently being "consumed" in subsequent computations. These procedure calls can be seen as the stages of a *pipelined* computation [14, Ch. 9].

In this work, we present our implementation of *multivariate power series* (Section 3) and *univariate polynomials over multivariate power series* "UPoPS" (Section 4) based on the ideas of lazy evaluation. Factoring such polynomials, by means of Hensel's lemma and its extensions and variants, like the extended Hensel construction (EHC) [17, 1] and the Jung-Abhyankar Theorem [16], is our driving application. We discuss a lazy implementation of factoring via Hensel's lemma (Section 6) by means of lazy Weierstrass preparation (Section 5).

Our implementation is part of the Basic Polynomial Algebra Subprograms (BPAS) library [3], a free and open-source computer algebra library for polynomial algebra. The library's core, of which our power series and UPoPS are a part, is carefully implemented in C for performance. The library also has a C++ interface for better usability. Such an interface for power series is forthcoming. Our current implementation is both sequential and over the field of rational numbers. However, the BPAS library has the necessary infrastructure, in particular asynchronous generators, see [4], to take advantage of the concurrency opportunities (essentially pipelining) created by our design based on lazy evaluation.

Existing implementations of multivariate power series are also available in MAPLE's `PowerSeries`[1] library [12, 2] and SAGEMATH [19]. The former is similarly based on lazy evaluation, while the latter uses the truncated power series approach mentioned above. Our experimental results show that our implementation in BPAS outperforms its counterparts by several orders of magnitude.

Lazy evaluation in computer algebra has some history, see the work of Karczmarczuk [11] (discussing different mathematical objects with an infinite length) and the work of Monagan and Vrbik [15] (discussing sparse polynomial arithmetic). Lazy univariate power series, in particular, have been implemented by Burge and Watt [6] and by van der Hoeven [20]. However, up to our knowledge, our implementation is the first for *multivariate* power series in a compiled code.

[1]This library is accessible, yet undocumented, in MAPLE 2020 as `RegularChains:-PowerSeries`. See `www.regularchains.org/documentation.html`

2 Background

This section gathers basic concepts about multivariate formal power series. We suggest the book of G. Fischer [9] for an introduction to the subject. We start with formal power series arithmetic. Let $\mathbb{K}$ be an algebraic number field and $\overline{\mathbb{K}}$ its algebraic closure. We denote by $\mathbb{K}[[X_1, \ldots, X_n]]$ the ring of formal power series with coefficients in $\mathbb{K}$ and with variables $X_1, \ldots, X_n$.

Let $f = \sum_{e \in \mathbb{N}^n} a_e X^e$ be a formal power series and $d \in \mathbb{N}$. The *homogeneous part* and *polynomial part* of f in degree d are denoted by $f_{(d)}$ and $f^{(d)}$, and defined by $f_{(d)} = \sum_{|e|=d} a_e X^e$ and $f^{(d)} = \sum_{k \leq d} f_{(k)}$. Note that $e = (e_1, \ldots, e_n)$ is a multi-index, X^e stands for $X_1^{e_1} \cdots X_n^{e_n}$, $|e| = e_1 + \cdots + e_n$, and $a_e \in \mathbb{K}$ holds.

Let $f, g \in \mathbb{K}[[X_1, \ldots, X_n]]$. Then the *sum*, *difference*, and *product* of f and g are given by $f \pm g = \sum_{d \in \mathbb{N}} (f_{(d)} \pm g_{(d)})$ and $fg = \sum_{d \in \mathbb{N}} \left(\Sigma_{k+\ell=d} (f_{(k)} g_{(\ell)}) \right)$. The *order* of a formal power series $f \in \mathbb{K}[[X_1, \ldots, X_n]]$, denoted by $\mathrm{ord}(f)$, is defined as $\min\{d \mid f_{(d)} \neq 0\}$, if $f \neq 0$, and as ∞ otherwise. We recall several properties. First, $\mathbb{K}[[X_1, \ldots, X_n]]$ is an integral domain. Second, the set $\mathcal{M} = \{f \in \mathbb{K}[[X_1, \ldots, X_n]] \mid \mathrm{ord}(f) \geq 1\}$ is the only maximal ideal of $\mathbb{K}[[X_1, \ldots, X_n]]$. Third, for all $k \in \mathbb{N}$, we have $\mathcal{M}^k = \{f \in \mathbb{K}[[X_1, \ldots, X_n]] \mid \mathrm{ord}(f) \geq k\}$.

Krull topology. Let $(f_n)_{n \in \mathbb{N}}$ be a sequence of elements of $\mathbb{K}[[X_1, \ldots, X_n]]$ and $f \in \mathbb{K}[[X_1, \ldots, X_n]]$. We say that $(f_n)_{n \in \mathbb{N}}$ *converges* to f if for all $k \in \mathbb{N}$ there exists $N \in \mathbb{N}$ s.t. for all $n \in \mathbb{N}$ we have $n \geq N \Rightarrow f - f_n \in \mathcal{M}^k$. We say that $(f_n)_{n \in \mathbb{N}}$ is a *Cauchy sequence* if for all $k \in \mathbb{N}$ there exists $N \in \mathbb{N}$ s.t. for all $n, m \in \mathbb{N}$ we have $n, m \geq N \Rightarrow f_m - f_n \in \mathcal{M}^k$. The following results hold: we have $\bigcap_{k \in \mathbb{N}} \mathcal{M}^k = \langle 0 \rangle$. Moreover, if every Cauchy sequence in $\mathbb{K}$ converges, then every Cauchy sequence of $\mathbb{K}[[X_1, \ldots, X_n]]$ converges too.

Inverse of a power series. Let $f \in \mathbb{K}[[X_1, \ldots, X_n]]$. Then, the following properties are equivalent: (i) f is a unit, (ii) $\mathrm{ord}(f) = 0$, (iii) $f \notin \mathcal{M}$. Moreover, if f is a unit, then the sequence $(u_n)_{n \in \mathbb{N}}$, where $u_n = 1 + g + g^2 + \cdots + g^n$ and $g = 1 - f/f_{(0)}$, converges to the inverse of $f/f_{(0)}$.

Assume $n \geq 1$. Denote by $\mathbb{A}$ the ring $\mathbb{K}[[X_1, \ldots, X_{n-1}]]$ and by $\mathcal{M}$ be the maximal ideal of $\mathbb{A}$. Note that $n = 1$ implies $\mathcal{M} = \langle 0 \rangle$.

Lemma 1 *Let $f, g, h \in \mathbb{A}$ such that $f = gh$ holds. Assume $n \geq 2$. We write $f = \sum_{i=0}^{\infty} f_i$, $g = \sum_{i=0}^{\infty} g_i$ and $h = \sum_{i=0}^{\infty} h_i$, where $f_i, g_i, h_i \in \mathcal{M}^i \setminus \mathcal{M}^{i+1}$ holds for all $i > 0$, with $f_0, g_0, h_0 \in \mathbb{K}$. We note that these decompositions are uniquely defined. Let $r \in \mathbb{N}$. We assume that $f_0 = 0$ and $h_0 \neq 0$ both hold. Then the term g_r is uniquely determined by $f_1, \ldots, f_r, h_0, \ldots, h_{r-1}$.*

Lemma 1 is essential to our implementation of Weierstrass Preparation Theorem (WPT). Hence, we give a proof by induction on r. Since $g_0 h_0 = f_0 = 0$ and $h_0 \neq 0$ both hold, the claim is true for $r = 0$. Now, let $r > 0$ and we can assume that $g_0, \ldots, g_{r-1}$ are uniquely determined by $f_1, \ldots, f_{r-1}, h_0, \ldots, h_{r-2}$. Observe that to determine g_r, it suffices to expand $f = gh$ modulo $\mathcal{M}^{r+1}$:

$$f_1 + f_2 + \cdots + f_r = g_1 h_0 + (g_2 h_0 + g_1 h_1) + \cdots + (g_r h_0 + g_{r-1} h_1 + \cdots + g_1 h_{r-1}).$$

g_r is then found by polynomial multiplication and addition and a division by h_0.

Now, let $f \in \mathbb{A}[[X_n]]$, written as $f = \sum_{i=0}^{\infty} a_i X_n^i$ with $a_i \in \mathbb{A}$ for all $i \in N$. We assume $f \not\equiv 0 \mod \mathcal{M}[[X_n]]$. Let $d \geq 0$ be the smallest integer such that $a_d \notin \mathcal{M}$.

Then, WPT states the following.

Theorem 1 *There exists a unique pair (α, p) satisfying the following:*

(i) *α is an invertible power series of $\mathbb{A}[[X_n]]$,*

(ii) *$p \in \mathbb{A}[X_n]$ is a monic polynomial of degree d,*

(iii) *writing $p = X_n^d + b_{d-1}X_n^{d-1} + \cdots + b_1X_n + b_0$, we have: $b_{d-1}, \ldots, b_1, b_0 \in \mathcal{M}$,*

(iv) *$f = \alpha p$ holds.*

Moreover, if f is a polynomial of $\mathbb{A}[X_n]$ of degree $d + m$, for some m, then α is a polynomial of $\mathbb{A}[X_n]$ of degree m.

Proof. If $n = 1$, then writing $f = \alpha X_n^d$ with $\alpha = \sum_{i=0}^{\infty} a_{i+d}X_n^i$ proves the existence of the claimed decomposition. Now assume $n \geq 2$. Let us write $\alpha = \sum_{i=0}^{\infty} c_iX_n^i$ with $c_i \in \mathbb{A}$ for all $i \in \mathbb{N}$. Since we require α to be a unit, we have $c_0 \notin \mathcal{M}$. We must then solve for $b_{d-1}, \ldots, b_1, b_0, c_0, c_1, \ldots, c_d, \ldots$ such that for all $m \geq 0$ we have:

$$\begin{array}{rcl} a_0 & = & b_0c_0 \\ a_1 & = & b_0c_1 + b_1c_0 \\ a_2 & = & b_0c_2 + b_1c_1 + b_2c_0 \\ & \vdots & \\ a_{d-1} & = & b_0c_{d-1} + b_1c_{d-2} + \cdots + \cdots + b_{d-2}c_1 + b_{d-1}c_0 \\ a_d & = & b_0c_d + b_1c_{d-1} + \cdots + \cdots + b_{d-1}c_1 + c_0 \\ a_{d+1} & = & b_0c_{d+1} + b_1c_d + \cdots + \cdots + b_{d-1}c_2 + c_1 \\ & \vdots & \\ a_{d+m} & = & b_0c_{d+m} + b_1c_{d+m-1} + \cdots + \cdots + b_{d-1}c_{m+1} + c_m \\ & \vdots & \end{array}$$

We will compute each of $b_{d-1}, \ldots, b_1, b_0, c_0, c_1, \ldots, c_d, \ldots$ modulo each of the successive powers of $\mathcal{M}$, that is, $\mathcal{M}, \mathcal{M}^2, \ldots, \mathcal{M}^r, \ldots$. We start modulo $\mathcal{M}$. By definition of d, the left hand sides of the first d equations above are all $0 \mod \mathcal{M}$. Since c_0 is a unit, each of $b_0, b_1, \ldots, b_{d-1}$ is $0 \mod \mathcal{M}$. Plugging this into the remaining equations we obtain $c_i \equiv a_{d+i} \mod \mathcal{M}$, for all $i \geq 0$. Therefore, we have solved for each of $b_{d-1}, \ldots, b_1, b_0, c_0, c_1, \ldots, c_d, \ldots$ modulo $\mathcal{M}$. Let $r > 0$ be an integer. We assume that we have inductively determined each of $b_{d-1}, \ldots, b_1, b_0, c_0, c_1, \ldots, c_d, \ldots$ modulo each of $\mathcal{M}, \ldots, \mathcal{M}^r$. We wish to determine them modulo $\mathcal{M}^{r+1}$. Consider the first equation, namely $a_0 = b_0c_0$, with $a_0, b_0, c_0 \in \mathbb{A}$. It follows from the hypothesis and Lemma 1 that we can compute b_0 modulo $\mathcal{M}^{r+1}$. Consider the second equation, that we re-write $a_1 - b_0c_1 = b_1c_0$. A similar reasoning applies and we can compute b_1 modulo $\mathcal{M}^{r+1}$. Continuing in this manner, we can compute each of $b_2, \ldots, b_{d-1}$ modulo $\mathcal{M}^{r+1}$. Finally, using the remaining equations, determine $c_i \mod \mathcal{M}^{r+1}$, for all $i \geq 0$. □

This theorem allows for three remarks. First, the assumption of the theorem, namely $f \not\equiv 0 \mod \mathcal{M}[[X_n]]$, can always be met, for any $f \neq 0$, by a suitable linear change of coordinates. Second, WPT can be used to prove that $\mathbb{K}[[X_1, \ldots, X_n]]$ is both a unique factorization domain (UFD) and a Noetherian ring. Third, in the context of the theory of analytic functions, WPT implies that any analytic function (namely f in our context) resembles a polynomial (namely p in our context) in the vicinity of the origin.

Now, let $f = a_k Y^k + \cdots + a_1 Y + a_0$ with $a_k, \ldots, a_0 \in \mathbb{K}[[X_1, \ldots, X_n]]$. We define $\overline{f} = f(0, \ldots, 0, Y) \in \mathbb{K}[Y]$. We assume that f is monic in Y ($a_k = 1$). We further assume $\mathbb{K}$ is algebraically closed. Thus, there exist positive integers $k_1, \ldots, k_r$ and pairwise distinct elements $c_1, \ldots, c_r \in \mathbb{K}$ such that we have $\overline{f} = (Y - c_1)^{k_1} (Y - c_2)^{k_2} \cdots (Y - c_r)^{k_r}$.

Theorem 2 (Hensel's Lemma) *There exists $f_1, \ldots, f_r \in \mathbb{K}[[X_1, \ldots, X_n]][Y]$, all monic in Y, such that we have:*

1. $f = f_1 \cdots f_r$,
2. $\deg(f_j, Y) = k_j$, *for all* $j = 1, \ldots, r$,
3. $\overline{f_j} = (Y - c_j)^{k_j}$, *for all* $j = 1, \ldots, r$.

Proof. The proof is by induction on r. Assume first $r = 1$. Observe that $k = k_1$ necessarily holds. Now define $f_1 := f$. Clearly f_1 has all the required properties. Assume next $r > 1$. We apply a change of coordinates sending c_r to 0. That is: $g(X_1, \ldots, X_n, Y) := f(X_1, \ldots, X_n, Y + c_r) = (Y + c_r)^k + a_1 (Y + c_r)^{k-1} + \cdots + a_k$. WPT applies to g. Hence there exist $\alpha, p \in \mathbb{K}[[X_1, \ldots, X_n]][Y]$ such that α is a unit, p is a monic polynomial of degree k_r, with $\overline{p} = Y^{k_r}$, and we have $g = \alpha p$. Then, we set $f_r(Y) = p(Y - c_r)$ and $f^* = \alpha(Y - c_r)$. Thus f_r is monic in Y and we have $f = f^* f_r$. Moreover, we have $\overline{f^*} = (Y - c_1)^{k_1} (Y - c_2)^{k_2} \cdots (Y - c_{r-1})^{k_{r-1}}$. The induction hypothesis applied to f^* implies the existence of $f_1, \ldots, f_{r-1}$. $\square$

3 The Design and Implementation of Lazy Power Series

Our power series implementation is both lazy and high-performing. To achieve this, our design and implementation has two goals:
(i) compute only terms of the series which are truly needed; and
(ii) have the ability to "resume" a computation, in order to obtain a higher precision power series without restarting from the beginning.
Of course, the lazy nature of our implementation refers directly to (i), while the high-performance nature is due in part to (ii) and in part to other particular implementation details to be discussed.

Facilitating both of these aspects requires the use of some sort of generator function—a function which returns new terms for a power series to increase its precision. Such a *generator*, is the key to high-performance in our implementation, yet also the most difficult part of the design.

Our goal is to define a structure encoding power series so that they may be dynamically updated on request. Each power series could then be represented as a polynomial alongside some generator function. A key element of this design is to "hide" the updating of the underlying polynomial. In our C implementation this is done through a functional interface comprising of two main functions: (i) getting the homogeneous part of a power series, and (ii) getting the polynomial part of

```
geometric_series_ps := proc(vars::list)
    local homog_parts := proc(vars::list)
        return d -> sum(vars[i], i=1..nops(vars))^d;
    end proc;
    ps := table();
    ps[DEG] := 0;
    ps[GEN] := homog_parts(vars); #capture vars in closure, return a function
    ps[POLY] := ps[GEN](0);
    return ps;
end proc;
```

Listing 1: The geometric series as a lazy power series.

a power series, each for a requested degree. These functions call some underlying generator to produce terms until the requested degree is satisfied.

As a first example, consider, the construction of the geometric series as a lazy power series, in MAPLE-style pseudo-code, in Listing 1. A power series is a data structure holding a polynomial, a generator function, and an integer to indicate up to which degree the power series is currently known. In this simple example, we see the need to treat functions as first-class objects. The manipulation of such functions is easy in functional or scripting languages, where dynamic typing and first-class function objects support such manipulation. This manipulation becomes further interesting where the generator of a power series must invoke other generators, as in the case of arithmetic (see Section 3.2).

In support of high-performance we choose to implement our power series in the strongly-typed and compiled C programming language rather than a scripting language. On one hand, this allows direct access to our underlying high-performance polynomial implementation [5], but on the other hand creates an impressive design challenge to effectively handle the need for dynamic function manipulation. In this section we detail our resulting solution, which makes use of a so-called *ancestry* in order for the generator function of a newly created power series to "remember" from where it came. We begin by discussing the power series data structure, and our solution to generator functions in C. Then, Section 3.2 examines power series multiplication and division using this structure, and evaluates our arithmetic performance against SAGEMATH and MAPLE.

3.1 The Power Series Data Structure, Generators, and Ancestors

The organization of our power series data structure is focused on supporting incremental generation of new terms through continual updates. To support this, the first fundamental design element is the storage of terms of the power series. The current polynomial part, i.e. the terms computed so far, of a power series are stored in a *graded representation*. A dense array of (pointers to) polynomials is maintained whereby the index of a polynomial in this array is equal to its (total) degree. Thus, this is an array of homogeneous polynomials representing the homogeneous parts of the power series, called the *homogeneous part array*. The power series data structure is a simple C struct holding this array, as well as integer numbers indicating the degree up to which homogeneous parts are currently known, and the allocation size of the homogeneous part array.

Using our graded representation, the generator function is simply a function returning the homogeneous part of a power series for a requested degree. Unfortunately, in the C language, functions are not readily handled as objects. Hence, we look to essentially create a *closure* for the generator function (see, e.g., [18, Ch. 3]), by storing a function pointer along with the values necessary for the function. For simplicity of implementation, these captured values are passed to the function as arguments. We first describe this function pointer.

In an attempt to keep the generators as simple as possible, we enforce some symmetry between all generators and thus the stored function pointers. Namely: (*i*) the first parameter of each generator must be an integer, indicating the degree of the homogeneous polynomial to be generated; and (*ii*) they must return that homogeneous polynomial. For some generator functions, e.g. the geometric series, this single integer argument is enough to obtain a particular homogeneous part. However, this is insufficient for most cases, particularly for generating a homogeneous part of a power series which resulted from an arithmetic operation.

Therefore, to introduce some flexibility in the generators, we extend the previous definition of a generator function to include a finite number of `void` pointer parameters following the first integer parameter. The use of `void` pointer parameters is a result of the fact that function pointers must be declared to point to a function with a particular number and type of parameters. Since we want to store this function pointer in the power series struct, we would otherwise need to capture all possible function declarations, which is a very rigid solution. Instead, `void` pointer parameters simultaneously allow for flexibility in the types of the generator parameters, as well as limit the number of function pointer types which must be captured by the power series struct. Flexibility arises where these `void` pointers can be cast to any other pointer type, or even cast to any machine-word-sized plain data type (e.g. `long` or `double`). In our implementation these so-called *void generators* are simple wrappers, casting each `void` pointer to the correct data type for the particular generator, and then calling the *true generator*. Section 3.2 provides an example in Listing 4.

Our implementation, which supports power series arithmetic, Weierstrass preparation, and factorization via Hensel's lemma, currently requires only 4 unique types of function pointers for these generators. All of these function pointers return a polynomial and take an integer as the first parameter. They differ in taking 0–3 `void` pointer parameters as the remaining parameters. We call the number of these `void` pointer parameter the generator's *order*. We then create a union type for these 4 possible function pointers and store only the union in the power series struct. The generator's order is also stored as an integer to be able to choose the correct generator from the union type at runtime.

Finally, these `void` pointers are also stored in the struct to eventually be passed to the generator. When the generator's order is less than maximum, these extra `void` pointers are simply set to `NULL`. The structure of these generators, the generator union type, and the power series struct itself is shown in Listing 2. In our implementation, these generators are used generically, via the aforementioned functional interface. In the code listings which follow, these functions are named `homogPart_PS` and `polynomialPart_PS`, to compute the homogeneous part and polynomial part of a power series, respectively.

In general, these void pointer generator parameters are actually pointers to

```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_tertiary)(int, void*, void*, void*);

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

typedef struct PowerSeries {
    int deg;
    int alloc;
    Poly_ptr* homog_polys;
    HomogPartGenerator_u gen;
    int genOrder;
    void *genParam1, *genParam2, *genParam3;
} PowerSeries_t;
```

Listing 2: A first implementation of the power series struct in C and function pointer declarations for the possible generator functions. `Poly_ptr` is a pointer to a polynomial.

existing power series structs. For example, the operands of an arithmetic operation would become arguments to the generator of the result. This relation then yields a so-called *ancestry* of power series. In this indirect way, a power series "remembers" from where it came, in order to update itself upon request via its generator. This may trigger a cascade of updates where updating a power series requires updating its "parent" power series, and so on up the *ancestry tree*. Section 3.2 explores this detail in the context of power series arithmetic, meanwhile it is also discussed as a crucial part of a lazy implementation of Weierstrass preparation (Section 5) and factorization via Hensel's lemma (Section 6).

The implementation of this ancestry requires yet one more additional feature. Since our implementation is in the C language, we must manually manage memory. In particular, references to parent power series (via the void pointers) must remain valid despite actions from the user. Indeed, the underlying updating mechanism should be transparent to the end-user. Thus, it should be perfectly valid for an end-user to obtain, for example, a power series product, and then free the memory associated with the operands of the multiplication.

In support of this we have established a reference counting scheme. Whenever a power series is made the parent of another power series its reference count is incremented. Therefore, the user may "free" or "destroy" a power series when it is no longer needed, but the memory persists as long as some other power series has reference to it. Destruction is then only a decrement of a reference counter. However, once the counter falls to 0, the data is actually freed, and moreover, a child power series will decrement the reference count of its parents. In a final complication, we must consider the case when a void pointer parameter is not pointing to a power series. We resolve this by storing, in the power series struct, a value to identify the actual type of a void parameter. A simple `if` condition can then check this type and conditionally free the generator parameter, if it is not plain data. For example, a power series or a UPoPS, see Listing 3.

```c
typedef enum GenParamType {
    PLAIN_DATA = 0,
    POWER_SERIES = 1,
    UPOPS = 2,
    MPQ_LIST = 3
} GenParamType_e;

void destroyPowerSeries_PS(PowerSeries_t* ps) {
    --(ps->refCount);
    if (ps->refCount <= 0) {
        for (int i = 0; i <= ps->deg; ++i) {
        freePolynomial(homog_polys[i]);
        }
        if (ps->genParam1 != NULL && ps->paramType1 == POWER_SERIES) {
            destroyPowerSeries_PS((PowerSeries_t*) ps->genParam1);
        }
        // repeat for other parameters.
    }
}
```

Listing 3: Extending the power series struct to include reference counting and proper management of reference counts to parent power series via `destroyPowerSeries_PS`.

3.2 Implementing Power Series Arithmetic

With the power series structure fully defined, we are now able to see examples putting its generators to use. Given the design established in the previous section, implementing a power series operation is as simple as defining the unique generator associated with that operation. In this section we present power series multiplication and division using this design. Let us begin with the former.

As we have seen in Section 2, the power series product of $f, g \in \mathbb{K}[[X_1, \ldots, X_n]]$ is defined simply as $h = fg = \sum_{d\in\mathbb{N}} \left(\Sigma_{k+\ell=d} \left(f_{(k)} g_{(\ell)}\right)\right)$. In our graded representation, continually computing new terms of h requires simply computing homogeneous parts of increasing degree. Indeed, for a particular degree d we have $(fg)_{(d)} = \sum_{k+\ell=d} f_{(k)} g_{(\ell)}$. Through our use of an ancestry and generators, the power series h can be constructed lazily, by simply defining its generator and generator parameters, and instantly returning the resulting struct. The generator in this case is exactly a function to compute $(fg)_{(d)}$ from f and g.

In reality, the generator stored in the struct encoding h is the *void generator* `homogPartVoid_prod_PS` which, after casting parameters, simply calls the *true generator*, `homogeneousPart_prod_PS`. This is shown in Listing 4. There, `multiplyPowerSeries_PS` is the actual power series operator, returning a lazily constructed power series product. There, the parents `f` and `g` are reserved (reference count incremented) and assigned to be generator parameters, and the generator function pointer set. Finally, a single term of the product is computed.

Now consider finding the quotient $h = \sum_e c_e X^e$ which satisfies $f = gh$ for a given power series $f = \sum_e a_e X^e$ and an invertible power series $g = \sum_e b_e X^e$. One could proceed by equating coefficients in $f = gh$, with b_0 being the constant term of g, to obtain $c_e = {}^1/_{b_0} \left(a_e - \sum_{k+\ell=e} b_k c_\ell\right)$. This formula can easily be rearranged in order to find the homogeneous part of h for a given degree d: $h_{(d)} = {}^1/_{g_{(0)}} \left(f_{(d)} - \sum_{k=1}^{d} g_{(k)} h_{(d-k)}\right)$. This formula is possible since to compute $h_{(d)}$ we need only $h_{(i)}$ for $i = 1, \ldots, d-1$. Moreover, the base case is simply $h_{(0)} = f_{(0)}/g_{(0)}$, a valid division in $\mathbb{K}$ since $g_{(0)} \neq 0$. The rest follows by induction.

```c
Poly_ptr homogPart_prod_PS(int d, PowerSeries_t* f, PowerSeries_t* g) {
    Poly_ptr sum = zeroPolynomial();
    for (int i = 0; i <= d; i++) {
        Poly_ptr prod = multiplyPolynomials(
                             homogPart_PS(d-i, f),  homogPart_PS(i, g));
        sum = addPolynomials(sum, prod);
    }
    return sum;
}

Poly_ptr homogPartVoid_prod_PS(int d, void* param1, void* param2) {
    return homogPart_prod_PS(d, (PowerSeries_t*) param1,
                                (PowerSeries_t*) param2);
}

PowerSeries_t* multiplyPowerSeries_PS(PowerSeries_t* f, PowerSeries_t* g) {
    if (isZeroPowerSeries_PS(f) || isZeroPowerSeries_PS(g)) {
        return zeroPowerSeries_PS();
    }
    reserve_PS(f); reserve_PS(g);
    PowerSeries_t* prod = allocPowerSeries(1);
    prod->gen.binaryGen= &(homogPartVoid_prod_PS)
    prod->genParam1 = (void*) f;
    prod->genParam2 = (void*) g;
    prod->paramType1 = POWER_SERIES;
    prod->paramType2 = POWER_SERIES;
    prod->deg = 0;
    prod->homogPolys[0] = homogPart_prod_PS(0, f, g);
    return prod;
}
```

Listing 4: Computing the multiplication of two power series, where `homogPart_prod_PS` is the generator of the product.

In our graded representation, this formula yields a generator for a power series quotient. The realization of this generator in code is simple, as shown in Listing 5. Not shown is the void generator wrapper and a top-level function to return the lazy quotient, which is simply symmetric to the previous multiplication example.

The only trick to this generator for the quotient is that it requires a reference to the quotient itself. This creates an issue of a circular reference in the power series ancestry. To avoid this, we abuse our parameter typing and label the quotient's reference to itself as plain data.

We now look to compare our implementation against SageMath [19], and Maple 2020. In Maple, the `PowerSeries` library [12, 2] provides lazy multivariate power series, meanwhile the built-in `mtaylor` command provides *truncated* multivariate taylor series. Similarly, SageMath includes only truncated power series. In these latter two, an explicit precision must be used and truncations cannot be extended once computed. Consequently, our experimentation only measures computing a particular precision, thus not using our implementation's ability to resume computation. We compare against all three; see Figures 1–3.

In SageMath, the multivariate power series ring $R[[X_1, \ldots, X_n]]$ is implemented using the univariate power series ring $S[[T]]$ with $S = R[X_1, \ldots, X_n]$. In $S[[T]]$, the subring formed by all power series f such that the coefficient of T^i in f is a homogeneous polynomial of degree i (for all $i \geq 0$) is isomorphic to $R[[X_1, \ldots, X_n]]$. By default, Singular [8] underlies the multivariate polynomial ring S while Flint [10] underlies the univariate polynomials used in univariate power series. Python 3.7 interfaces and joins these underlying implementations. To see ex-

```
Poly_ptr homogPart_quo_PS(int d, PowerSeries_t* f, PowerSeries_t* g,
     PowerSeries_t* h) {
    if (d == 0) {
        return dividePolynomials(homogPart_PS(0, f), homogPart_PS(0, g));
    }
    Poly_ptr s = homogPart_PS(d, f);
    for (int i = 1; i <= deg; ++i) {
        Poly_ptr p = multiplyPolynomials(homogPart_PS(i, g),
                                         homogPart_PS(d-i, h));
        s = subPolynomials(s, p);
    }
    return divideByRational(s, homogPart(0, g))
}
```

Listing 5: Computing the division of two power series, where `homogPart_quo_PS` is the genrator of the quotient.

actly how SageMath works consider $f \in \mathbb{K}[[X_1, X_2]]$ with the goal is to compute $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ to precision d. One begins by constructing the power series ring in X_1, X_2 over $\mathbb{Q}$ with the default precision set to d as `R.<x,y> = PowerSeriesRing(QQ, default_prec=d)`. Then `g = f^-1` returns the inverse, and `h = f * g` the desired product, to precision d.

Throughout this paper our benchmarks were collected with a time limit of 1800 seconds on a machine running Ubuntu 18.04.4 with an Intel Xeon X5650 processor running at 2.67 GHz, with 12x4GB DDR3 memory at 1.33 GHz.

The first set of benchmarks are presented in Figure 1 where the power series $f = 1 + X_1 + X_2$ is both inverted and multiplied by its inverse. Figures 2 and 3 present the same but for $f = 1+X_1+X_2+X_3$ and $f = 2+\frac{1}{3}(X_1+X_2)$, respectively. In all cases, $f \cdot \frac{1}{f}$ includes the time to compute the inverse. It is clear that our implementation is orders of magnitude faster than existing implementations. This is due in part to the efficiency of our underlying polynomial arithmetic implementation [5], but also to our execution environment. Our implementation is written in the C language and fully compiled, meanwhile, both SageMath and Maple have a level of interpreted code, which surely impacts performance. We note that, through truncated power series as polynomials, the dense multiplication of a power series by its inverse is trivial for SageMath and `mtaylor`.

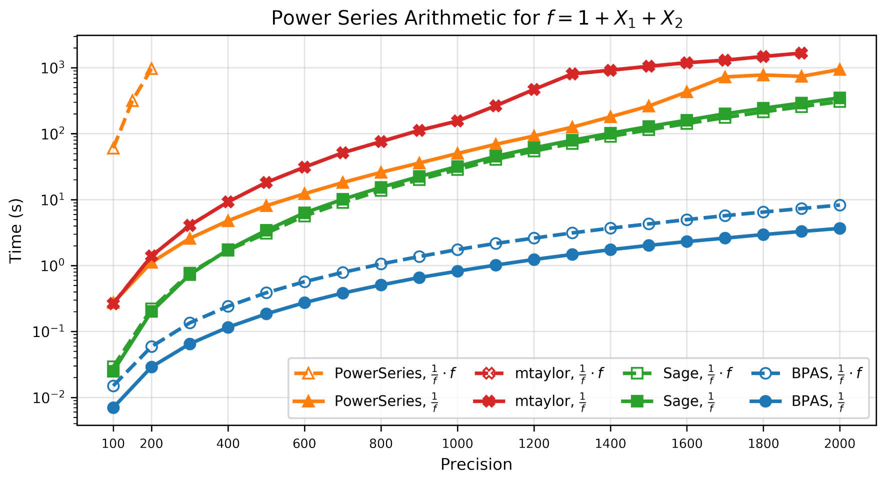

Figure 1: Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 1 + X_1 + X_2$

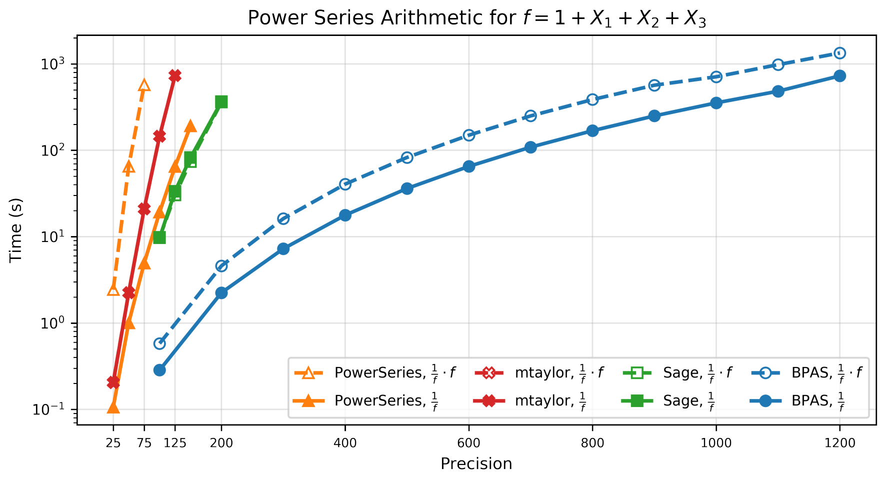

Figure 2: Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 1 + X_1 + X_2 + X_3$.

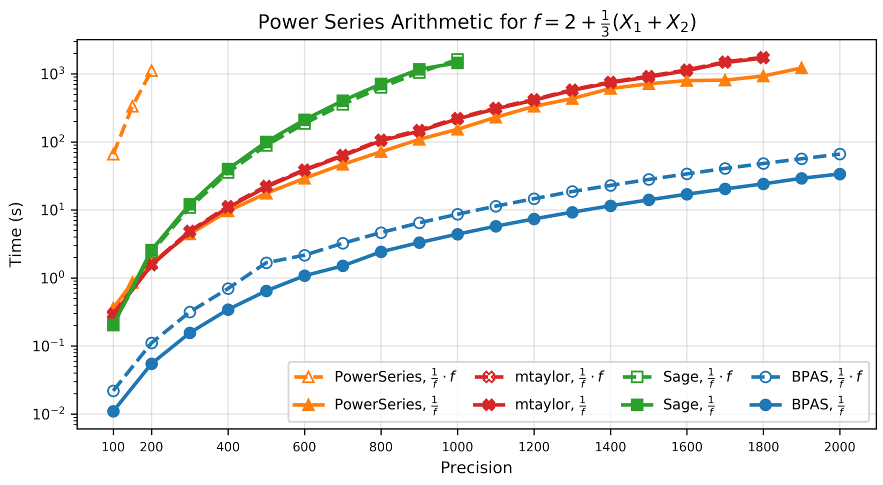

Figure 3: Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 2 + \frac{1}{3}(X_1 + X_2)$

4 Univariate Polynomials over Lazy Power Series

A univariate polynomial with multivariate power series coefficients, i.e. a univariate polynomial over power series (UPoPS), is implemented as a simple extension of our existing power series. Following a simple dense univariate polynomial design, our UPoPS are represented as an array of coefficients, each being a pointer to a power series, where the index of the coefficient in the array implies the degree of the coefficient's associated monomial. Integers are also stored for the degree of the polynomial and the allocation size of the coefficient array. In support of the underlying lazy power series, we also add reference counting to UPoPS.

The arithmetic of UPoPS is inherited directly from its coefficient ring (our lazy power series) and follows a naive implementation of univariate polynomials (see, e.g. [21, Ch. 2]). Through the use of our lazy power series, our implementation of UPoPS is automatically lazy through each individual coefficient's ancestry. Lazy UPoPS addition, subtraction, and multiplication follow easily.

One important operation on UPoPS which is not inherited directly from our power series implementation is Taylor shift. This operation takes a UPoPS $f \in \mathbb{K}[[X_1,\ldots,X_n]][Y]$ and returns $f(Y+c)$ for some $c \in \mathbb{K}$. Normally, the shift operator would be defined for any element of the ground ring $\mathbb{K}[[X_1,\ldots,X_n]]$, however our use of Taylor shift in applying Hensel's lemma (see Section 6), requires only shifting by elements of $\mathbb{K}$, and we thus specialize to that case. Since the coefficients of f are lazy power series, our goal is to compute $f(Y+c)$ lazily as well. Since our UPoPS are represented in a dense fashion, we compute the coefficients of $f(Y+c)$ as a polynomial in Y. Let $\mathbf{S} = (s_{i,j})$ be the lower triangular matrix such that $s_{i,j}$ is the coefficient of Y^j in the binomial expansion $(Y+c)^i$, for $i = 0,\ldots,k$, and $j = 0,\ldots,i$, where $k = \deg(f)$. Let $\mathbf{A} = (a_i)$ be the vector of the coefficients of f and $\mathbf{B} = (b_i)$ that of the coefficients of $f(Y+c)$, so that we have $f(Y) = \sum_{0\le i\le k} a_i Y^i$ and $f(Y+c) = \sum_{0\le i\le k} b_i Y^i$. Then we can verify that b_i is the inner product of the i-th sub-diagonal of $\mathbf{S}$ with the lower $k+1-i$ elements of $\mathbf{A}$, for $i = 0,\ldots,k$. In particular for $i = 0$, the coefficient b_0 is the inner product of the diagonal of $\mathbf{S}$ and the vector $\mathbf{A}$.

Recalling that $c \in \mathbb{K}$, the construction of b_i can be performed in a graded fashion from the linear combinations of homogeneous parts of a_j for $j \le i$. The homogeneous part $b_{i_{(d)}}$ of degree d, can be computed from only $a_{j_{(d)}}$, for $j \le i$. Therefore, a generator for b_i is easily constructed from the homogeneous parts of a_j, for $j \le i$, using multiplication by elements of $\mathbb{K}$ and polynomial addition. Therefore, we can construct the entire UPoPS $f(Y+c)$ in a lazy manner through initializing each coefficient b_i with a so-called linear combination generator. Since the main application of Taylor shift is factorization via Hensel's lemma, we leave its evaluation to Section 6 where benchmarks for factorization are presented.

5 Lazy Weierstrass Preparation

In this section we consider the application of Weierstrass Preparation Theorem (WPT) to univariate polynomials over power series. Let $f, p, \alpha \in \mathbb{K}[[X_1,\ldots,X_n]][Y]$ where $f = \sum_{i=0}^{d+m} a_i Y^i$, $p = Y^d + \sum_{i=0}^{d-1} b_i Y^i$, and $\alpha = \sum_{i=0}^{m} c_i Y^i$. From the proof of WPT (Theorem 1), we have that $f = \alpha p$ implies the following equalities:

$$\begin{array}{rcl}
a_0 & = & b_0c_0 \\
a_1 & = & b_0c_1 + b_1c_0 \\
 & \vdots & \\
a_{d-1} & = & b_0c_{d-1} + b_1c_{d-2} + \cdots + b_{d-2}c_1 + b_{d-1}c_0 \\
a_d & = & b_0c_d + b_1c_{d-1} + \cdots + b_{d-1}c_1 + c_0 \\
 & \vdots & \\
a_{d+m-1} & = & b_{d-1}c_m + c_{m-1} \\
a_{d+m} & = & c_m
\end{array} \tag{1}$$

Following the proof, we wish to solve these equations modulo successive powers of $\mathcal{M}$, the maximal ideal of $\mathbb{K}[[X_1,\ldots,X_n]]$. This implies that we will be iteratively updating each power series $b_0,\ldots,b_{d-1},c_0,\ldots,c_m$ by adding homogeneous polynomials of increasing degree, precisely as we have done for all lazy power series operations thus far. To solve these equations modulo $\mathcal{M}^{r+1}$, both the proof of WPT and the algorithm operates in two phases. First, the coefficients $b_0,\ldots,b_{d-1}$ of p are updated using the equations from a_0 to a_{d-1}, one after the other. Second, the coefficients $c_0,\ldots,c_m$ of α are updated.

Let us begin with the first phase. Rearranging the equations that express a_0 to a_{d-1} shows their successive dependency where b_{i-1} is needed for b_i:

$$\begin{array}{rcl}
a_0 & = & b_0c_0 \\
a_1 - b_0c_1 & = & b_1c_0 \\
a_2 - b_0c_2 - b_1c_1 & = & b_2c_0 \\
 & \vdots & \\
a_{d-1} - b_0c_{d-1} - b_1c_{d-2} + \cdots - b_{d-2}c_1 & = & b_{d-1}c_0
\end{array} \tag{2}$$

Consider that $b_0,\ldots,b_{d-1},c_0,\ldots,c_m$ are known modulo $\mathcal{M}^r$ and $a_0,\ldots,a_{d-1}$ are known modulo $\mathcal{M}^{r+1}$. Using Lemma 1 the first equation $a_0 = b_0c_0$ can then be solved for b_0 modulo $\mathcal{M}^{r+1}$. From there, the expression $a_1 - b_0c_1$ then becomes known modulo $\mathcal{M}^{r+1}$. Notice that the constant term of b_0 is 0 by definition, thus the product b_0c_1 is known modulo $\mathcal{M}^{r+1}$ as long as b_0 is known modulo $\mathcal{M}^{r+1}$. Therefore, the entire expression $a_1 - b_0c_1$ is known modulo $\mathcal{M}^{r+1}$ and Lemma 1 can be applied to solve for b_1 in the equation $a_1 - b_0c_1 = b_1c_0$. This argument follows for all equations, therefore solving for all $b_0,\ldots,b_{d-1}$ modulo $\mathcal{M}^{r+1}$.

In the second phase, we look to determine $c_0,\ldots,c_m$ modulo $\mathcal{M}^{r+1}$. Here, we have already computed $b_0,\ldots,b_{d-1}$ modulo $\mathcal{M}^{r+1}$. A rearrangement of the remaining equations of (1) shows that each c_i may be computed modulo $\mathcal{M}^{r+1}$:

$$\begin{array}{rcl}
c_m & = & a_{d+m} \\
c_{m-1} & = & a_{d+m-1} - b_{d-1}c_m \\
c_{m-2} & = & a_{d+m-2} - b_{d-2}c_m - b_{d-1}c_{m-1} \\
 & \vdots & \\
c_0 & = & a_d - b_0c_d - b_1c_{d-1} - \cdots - b_{d-1}c_1
\end{array} \tag{3}$$

Consider the second equation. Observe that a_{d+m-1} and b_{d-1} are known modulo $\mathcal{M}^{r+1}$ and that $b_{d-1} \in \mathcal{M}$ holds. Then, the product $b_{d-1}c_m$ is known modulo $\mathcal{M}^{r+1}$ and we deduce c_{m-1} modulo $\mathcal{M}^{r+1}$. The same follows for $c_{m-2},\ldots,c_0$.

With these two sets of re-arranged equations, we have seen how the coefficients of p and α can be updated modulo successive powers of $\mathcal{M}$. That is to say, how they can be updated by adding homogeneous parts of successive degrees. This design lends itself to be implemented as generator functions.

The first challenge to this design is that each power series coefficient of p is not independent, and must be updated in a particular order. Moreover, to generate homogeneous parts of degree d for the coefficients of p, the coefficients of α must also be updated to degree $d-1$. Therefore, it is a required side effect of each generator of $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ that all other power series are updated. To implement this, the generators of the power series of p are a mere wrapper of the same underlying updating function which updates all coefficients simultaneously. This so-called *Weierstrass update* follows two phases as just explained.

In the first phase one must use Lemma 1 to solve for the homogeneous part of degree r for each $b_0, \ldots, b_{d-1}$. To achieve this effectively, our implementation follows two key points. The first is an efficient implementation of Lemma 1 itself. Consider again the equations of Lemma 1 for $f = gh$ modulo $\mathcal{M}^{r+1}$:

$$\begin{aligned} f_{(1)} + f_{(2)} + \cdots + f_{(r)} &= (g_{(1)} + g_{(2)} + \cdots + g_{(r)})(h_{(0)} + h_{(1)} + \cdots + h_{(r)}) \\ &= \big(g_{(1)}h_{(0)}\big) + \big(g_{(2)}h_{(0)} + g_{(1)}h_{(1)}\big) + \cdots + \\ &\quad \big(g_{(r)}h_{(0)} + g_{(r-1)}h_{(1)} + \cdots + g_{(1)}h_{(r-1)}\big) . \end{aligned} \tag{4}$$

The goal is to obtain $g_{(r)}$. What one should realize is that computing $g_{(r)}$ requires only a fraction of this formula. In particular, we have

$$f_{(r)} = g_{(r)}h_{(0)} + g_{(r-1)}h_{(1)} + \cdots + g_{(1)}h_{(r-1)}, \tag{5}$$

and $g_{(r)}$ can be computed with simply polynomial addition and multiplication, followed by the division of a single element of $\mathbb{K}$, since $h_{(0)}$ has degree 0.

The second key point is that, in order to compute $g_{(r)}$, i.e. the homogeneous parts of degree r of $b_0, \ldots, b_{d-1}$, we must first find $f_{(r)}$, i.e. the homogeneous parts of degree r of $a_0, a_1 - b_0c_1$, $a_2 - b_0c_2 - b_1c_1$, etc. from (2). A nice result of our existing power series design is that we can define some lazy power series, say F_i, such that $F_i = a_i - \sum_{k=0}^{i} b_k c_{i-k}$. These F_i can then be automatically updated via its generators when the b_k are updated. The implementation of phase one of Weierstrass update is then simply a loop over solving equation (5), where $f_{(r)}$ is automatically obtained through the use of generators on the power series F_i.

Phase two of Weierstrass update follows the same design as in the definition of those F_i power series. In particular, from (3) we can see that each $c_m, \ldots, c_0$ is merely the result of some power series arithmetic. Hence, we simply rely on the underlying power series arithmetic generators to be the generators of $c_m, \ldots, c_0$.

With the above discussion, we have fully defined a lazy implementation of Weierstrass preparation. It begins with an initialization, which simply uses lazy power series arithmetic to create $F_0, \ldots, F_{d-1}, c_m, \ldots, c_0$, and initializes each $b_0, \ldots, b_{d-1}$ to 0. Then, the generators for $b_0, \ldots, b_{d-1}$ all call the same underlying Weierstrass update function. This function is shown in Algorithm 1, which is split into two phases as our discussion has suggested.

In our implementation, we store a pointer to the array of $F_0, \ldots, F_{d-1}$ in the UPoPS struct of p for ease of calling Weierstrass update. This, along with the

Algorithm 1 WeierstrassUpdate(f, p, α, $\mathcal{F}$)

Input: $f = \sum_{i=0}^{d+m} a_i Y^i$, $p = \sum_{i=0}^{d} b_i Y^i$, $\alpha = \sum_{i=0}^{m} c_i Y^i$, $a_i, b_i, c_i \in \mathbb{K}[[X_1, \ldots, X_n]]$ satisfying Theorem 1, $\mathcal{F} = \{F_i \mid F_i = a_i - \sum_{k=0}^{i} b_k c_{i-k}, i = 0, \ldots, d-1\}$, with $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ known modulo $\mathcal{M}^r$, the maximal ideal of $\mathbb{K}[[X_1, \ldots, X_n]]$.
Output: $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ known modulo $\mathcal{M}^{r+1}$, updated in-place.

```
   ▷ phase one
1: for $i = 0$ to $d-1$ do
2:     $s := 0$
3:     for $k = 1$ to $r-1$ do
4:         $s := s +$ homogPart_PS($r-k$, $b_i$) $\times$ homogPart_PS($k$, $c_0$)
5:     homogPart_PS($r$, $b_i$) := ( homogPart_PS($r$, $F_i$) $-$ $s$ ) / homogPart_PS(0, $c_0$)
   ▷ phase two
6: for $i = 0$ to $m$ do
7:     homogPart_PS($r$, $c_i$)                ▷ force an update of $c_i$ for next update.
```

circular references between coefficients of p and α, creates a delicate situation for reference counting. Readers may refer to our code in BPAS [3] for our solution.

Notice also that, although phase one requires updating each b_i in order from $i = 0$ to $d-1$, the same is not true for $c_0, \ldots, c_m$. This second phase is embarrassingly parallel. Structuring Weierstrass preparation as a lazy operation also naturally exposes further concurrency opportunities, such a parallel pipeline structure in the case of factorization via Hensel's lemma, see Section 6.

Finally, we report on experimental results for Weierstrass preparation against the `PowerSeries` library. We note that the latter is not a lazy implementation, returning only a truncated UPoPS. We have studied two families of examples:
(i) $\frac{1}{1+X_1+X_2} Y^k + Y^{k-1} + \cdots + Y^2 + X_2 Y + X_1$ and
(ii) $\frac{1}{1+X_1+X_2} Y^k + Y^{k-1} + \cdots + Y^{\lceil k/2 \rceil} + X_2 Y^{\lceil k/2 \rceil - 1} + \cdots + X_2 Y + X_1$
The first results in p of degree 2, while the second results in p of degree $\lceil k/2 \rceil$, thus emphasizing the performance of phase two and phase one of the algorithm, respectively. The results of this experiment are summarized in Figures 4 and 5. Not only is our implementation orders of magnitude faster than MAPLE, but the difference in computation time further increases with increasing precision (total degree in X_1, X_2). This can be attributed to our efficient underlying power series arithmetic, as well as our smart implementation of Lemma 1.

6 Lazy Factorization via Hensel's Lemma

In Section 2 we have seen the description of Hensel's lemma for univariate polynomial over power series. Specifically, that the proof by construction provides a mechanism to factor UPoPS. We now look to make that construction lazy.

Recall that the proof of Theorem 2 provides a mechanism to factor a UPoPS $f \in \mathbb{K}[[X_1, \ldots, X_n]][Y]$ into factors $f_1, \ldots, f_r$ based on Taylor shift and repeated applications of Weierstrass preparation. The construction begins by first factorizing the polynomial $\bar{f} = f(0, \ldots, 0, Y) \in \mathbb{K}[Y]$, obtained by evaluating all variables in power series coefficients to 0, into linear factors. This can be performed with a suitable (algebraic) factorization algorithm for $\mathbb{K}$. For simplicity of presentation, let us assume that $\bar{f}$ factorizes into linear factors over $\mathbb{K}$, thus returning a list of roots $c_1, \ldots, c_r \in \mathbb{K}$ with respective multiplicities $k_1, \ldots, k_r$. The construction then

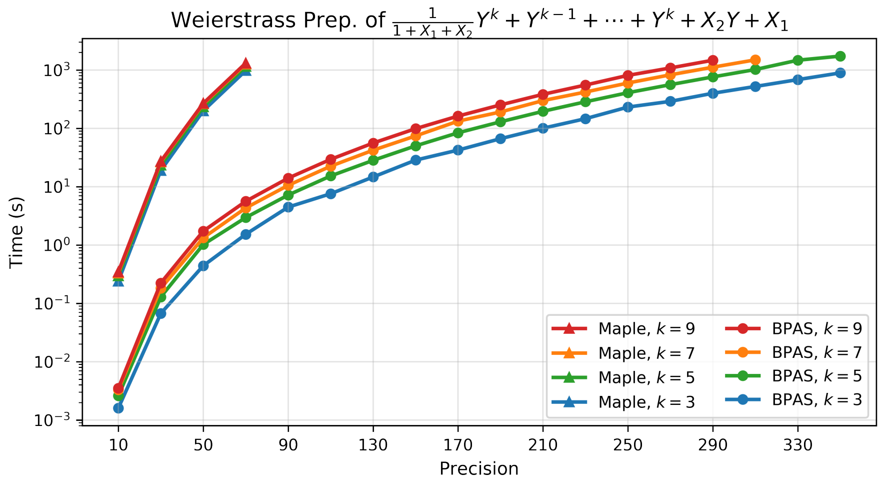

Figure 4: Applying Weierstrass preparation on family (i) for increasing precisions.

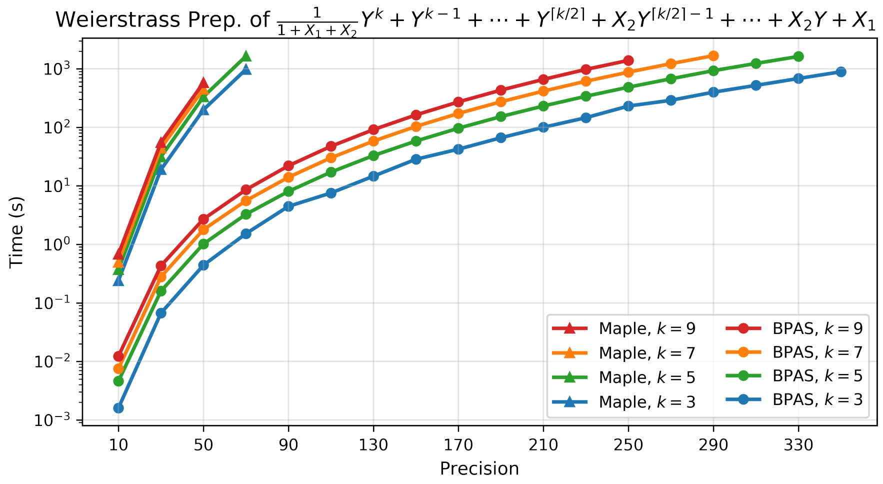

Figure 5: Applying Weierstrass preparation on family (ii) for increasing precisions.

proceeds recursively, obtaining one factor at a time.

Let us describe one step of the recursion, where f^* describes the current polynomial to factorize, initially being set to f. For a root c_i of $\bar{f}$, we perform a Taylor shift to obtain $g = f^*(Y + c_i)$ such that g has order k_i (as a polynomial in Y). The Weierstrass preparation theorem can then be applied to obtain p and $\alpha \in K[[X_1, \ldots, X_n]][Y]$ where p is monic and of degree k_i. A Taylor shift is then applied in reverse to obtain $f_i = p(Y - c_r)$, a factor of f, and $f^* = \alpha(Y - c_r)$, the UPoPS to factorize in the next step. The full procedure for obtaining all factors of f is shown as an iterative process, instead of recursive, in Algorithm 2.

The beauty of this algorithm is that it is immediately a lazy algorithm with no additional effort. Using the underlying lazy operations of Taylor shift (Section 4) and Weierstrass preparation (Section 5), the entire factorization is performed lazily, returning a factorization nearly instantly. The power series coefficients of these factors can automatically be updated later using their generators, which are simply

Algorithm 2 HenselFactorization(f)

```
Input: $f = \sum_{i=0}^{k} a_i Y^i, a_i \in \mathbb{K}[[X_1, \ldots, X_n]]$.
Output: $f_1, \ldots, f_r$ satisfying Theorem 2.
1: $\bar{f} = f(0, \ldots, 0, Y)$
2: $c_1, \ldots, c_r$ := obtain roots of $\bar{f}$        ▷ by some appropriate factorization algorithm
3: $f^* = f$
4: for $i = 1$ to $r$ do
5:     $g := f^*(Y + c_i)$
6:     $p, \alpha :=$ WeierstrassPreparation($g$)
7:     $f_i := p(Y - c_i)$
8:     $f^* := \alpha(Y - c_i)$
9: return $f_1, \ldots, f_r$
```

Taylor shift operations on top of a Weierstrass update.

Notice too the opportunities for concurrency exposed from a lazy Taylor shift and lazy Weierstrass. The factors $f_1, \ldots, f_r$ are created from successive applications of Weierstrass preparation. They in essence form a *pipeline* of processes [14, Ch. 9]. Updating one factor simultaneously causes its associated α from Weierstrass preparation to be updated. This in turn allows the next factor to be updated since this α is the input into the next Weierstrass preparation. This concurrency is on top of that available within a single Weierstrass preparation.

We now compare our implementation of factorization via Hensel's lemma in BPAS against that of MAPLE's `PowerSeries` library. In the latter, two functions are available for this operation: `ExtendedHenselConstruction` (EHC) and `FactorizationViaHenselLemma` (FVHL). FVHL has the same specifications as Algorithm 2 while EHC factorizes UPoPS over the field of Puiseux series in $X_1, \ldots, X_n$, see [1]. Our tests use two UPoPS f, one of degree 3 and one of degree 4, such that $\bar{f}$ splits into linear factors over $\mathbb{Q}$; in this way the output is the same for our BPAS code, EHC, and FVHL.

The results of this experiment are summarized in Figure 6 for the two UPoPS. Our implementation is orders of magnitude faster. We observe that the gap between our implementation and EHC increases both as UPoPS degree increases and as power series precision increases. A theoretical comparison, in terms of complexity analysis, between the EHC and Algorithm 2 is work in progress.

7 Conclusions and Future Work

Throughout this work we have explored the design and implementation of lazy multivariate power series, employing them in Weierstrass preparation and the factorization of univariate polynomials over power series via Hensel's lemma. Our implementation in the C language is orders of magnitude faster than existing implementations in SAGEMATH and MAPLE's `PowerSeries` library. In part, this is due to overcoming the challenge of working with dynamic generator functions in a compiled language, rather than using a more simplistic scripting language.

Yet, still more work can be done to further improve the performance of our implementation. The implementation of our arithmetic follows naive quadratic algorithms; instead, relaxed algorithms [20] should be integrated into our implementation to improve its algebraic complexity. Further, as mentioned in the case of Weierstrass preparation and in factorization via Hensel's lemma, there are opportu-

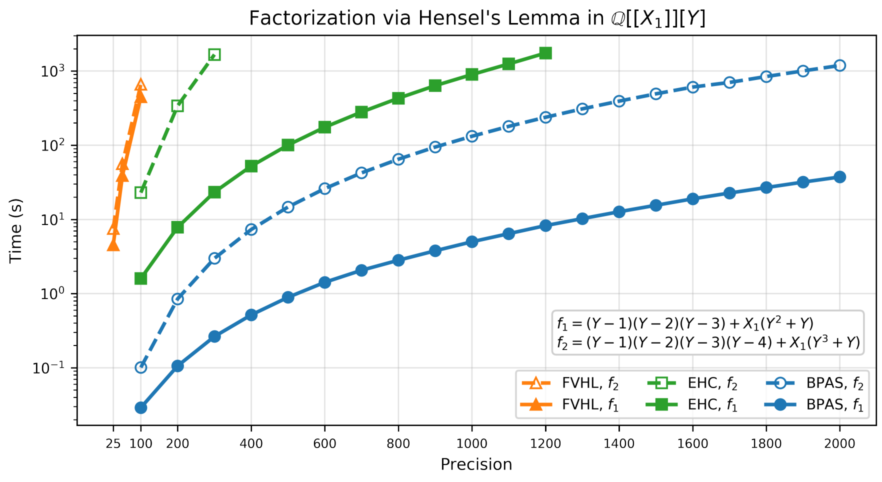

Figure 6: Applying factorization via Hensel's lemma to the UPoPS $f_1 = (Y-1)(Y-2)(Y-3) + X_1(Y^2+Y)$ and $f_2 = (Y-1)(Y-2)(Y-3)(Y-4) + X_1(Y^3+Y)$.

nities for concurrency in their implementation as lazy operations. This concurrency can be exploited with parallel programming techniques, including a parallel map and parallel pipeline, to yield further improved performance.

Acknowledgements

The authors would like to thank NSERC of Canada (award CGSD3-535362-2019) and Robert H. C. Moir.

References

[1] Parisa Alvandi, Masoud Ataei, Mahsa Kazemi, and Marc Moreno Maza. On the extended hensel construction and its application to the computation of real limit points. *J. Symb. Comput.*, 98:120–162, 2020.

[2] Parisa Alvandi, Mahsa Kazemi, and Marc Moreno Maza. Computing limits with the regularchains and powerseries libraries: from rational functions to zariski closure. *ACM Commun. Comput. Algebra*, 50(3):93–96, 2016.

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

[4] M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Yuzhen Xie. On the parallelization of triangular decomposition of polynomial systems. In *ISSAC 2020, Proceedings*, pages 22–29. ACM, 2020.

[5] Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, and Marc Moreno Maza. Algorithms and data structures for sparse polynomial arithmetic. *Mathematics*, 7(5):441, 2019.

[6] William H Burge and Stephen M Watt. Infinite structures in scratchpad ii. In *European Conference on Computer Algebra*, pages 138–148. Springer, 1987.

[7] Xavier Dahan, Marc Moreno Maza, Éric Schost, Wenyuan Wu, and Yuzhen Xie. Lifting techniques for triangular decompositions. In *ISSAC 2005, Beijing, China, 2005, Proceedings*, pages 108–115, 2005.

[8] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. Singular 4-1-1 — A computer algebra system for polynomial computations. `http://www.singular.uni-kl.de`, 2018.

[9] G. Fischer. *Plane Algebraic Curves.* AMS, 2001.

[10] W. Hart, F. Johansson, and S. Pancratz. FLINT: Fast Library for Number Theory, 2015. Version 2.5.2, `http://flintlib.org`.

[11] Jerzy Karczmarczuk. Generating power of lazy semantics. *Theor. Comput. Sci.*, 187(1-2):203–219, 1997.

[12] Mahsa Kazemi and Marc Moreno Maza. Detecting singularities using the powerseries library. In *Maple in Mathematics Education and Research - Third Maple Conference, MC 2019, Proceedings*, pages 145–155. Springer, 2019.

[13] M Lauer. Computing by homomorphic images. In *Computer Algebra*, pages 139–168. Springer, 1983.

[14] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation.* Elsevier, 2012.

[15] Michael B. Monagan and Paul Vrbik. Lazy and forgetful polynomial arithmetic and applications. In *CASC 2009, Proceedings*, pages 226–239, 2009.

[16] Adam Parusiski and Guillaume Rond. The AbhyankarJung theorem. *Journal of Algebra*, 365:29 – 41, 2012.

[17] T. Sasaki and F. Kako. Solving multivariate algebraic equation by Hensel construction. *Japan J. Indust. and Appl. Math.*, 1999.

[18] Michael L. Scott. *Programming Language Pragmatics (3. ed.).* Academic Press, 2009.

[19] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.1)*, 2020. `https://www.sagemath.org`.

[20] Joris van der Hoeven. Relax, but don't be too lazy. *J. Symb. Comput.*, 34(6):479–542, 2002.

[21] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, NY, USA, 2 edition, 2003.

[22] Joachim von zur Gathen. Hensel and Newton methods in valuation rings. *Mathematics of Computation*, 42(166):637–661, 1984.