Multivariate Power Series in Maple

Mohammadali Asadi\textsuperscript{1,2}, Alexander Brandt\textsuperscript{1}, Mahsa Kazemi\textsuperscript{1}, Marc Moreno Maza\textsuperscript{1}, and Erik Postma\textsuperscript{2}

\textsuperscript{1} The University of Western Ontario, 1151 Richmond St, London, Ontario, Canada
\textsuperscript{2} Maplesoft, 615 Kumpf Dr, Waterloo, Ontario, Canada

Abstract. We present \texttt{MultivariatePowerSeries}, a \textsc{maple} library introduced in \textsc{maple} 2021, providing a variety of methods to study formal multivariate power series and univariate polynomials over such series. This library offers a simple and easy-to-use user interface. Its implementation relies on lazy evaluation techniques and takes advantage of \textsc{maple}'s features for object-oriented programming. The exposed methods include Weierstrass Preparation Theorem and factorization via Hensel’s lemma. The computational performance is demonstrated by means of an experimental comparison with software counterparts.

Keywords: multivariate power series · Weierstrass Preparation Theorem · Hensel’s lemma · factorization · lazy evaluation

1 Introduction

In elementary courses on univariate calculus, power series are often introduced as limits of sequences of the form “the first $n$ terms of a given sequence”. This leads students to the study of analytic functions and the use of power series in computing function limits. While the extension of those notions to the multivariate case is a standard topic in advanced calculus courses, the availability of multivariate power series and multivariate analytic functions in computer algebra systems is somehow limited.

In \textsc{maple} \cite{maple}, \textsc{sagemath} \cite{sagemath}, and \textsc{mathematica} \cite{mathematica}, power series are restricted to being either only univariate or truncated, that is, reduced modulo a fixed power of the ideal $\langle X_1, \ldots, X_n \rangle$ generated by the variables of those power series. A truncated implementation, while simple, may be insufficient for, or computationally more expensive in, some particular circumstances. For instance, modern algorithms for polynomial system solving require the intensive use of modular methods based on Hensel lifting. In those lifting procedures, degrees of truncation may not be known a priori, thus leading to truncated power series being ineffective.

Considering that a power series has potentially an infinite number of terms naturally suggests to represent it as a procedure which, given a particular (total) degree, produces the terms of that degree. This leads to a so-called lazy evaluation scheme, where the terms of any power series are produced only as needed, via such a \textit{generator} function.
The usefulness of lazy evaluation in computer algebra has been studied for a few decades. In particular, see the work of Karczmarczuk [10], discussing different mathematical objects with an infinite length; Burge and Watt [7], and van der Hoeven [17], discussing lazy univariate power series; and Monagan and Vrbik [12], discussing lazy arithmetic for polynomials.

In this paper, we present MultivariatePowerSeries, which is among the new features released in MAPLE 2021 and publicly available in [1]. This library, written in the MAPLE language, provides the ability to create and manipulate multivariate power series with rational or algebraic number coefficients, as well as univariate polynomials whose coefficients are multivariate power series. Through lazy evaluation techniques and a careful implementation, our library achieves very high performance. These power series and univariate polynomials over power series (UPoPS) are employed in optimized implementations of Weierstrass Preparation Theorem and factorization of UPoPS via Hensel’s lemma.

Our implementation follows the lazy evaluation scheme of multivariate power series in the BPAS library [3]. The multivariate power series of BPAS, written in the C language, is discussed in [6] and extends upon the work of the PowerSeries subpackage of the RegularChains MAPLE library [2,13]. The PowerSeries package is the only preexisting implementation of multivariate power series integrated in MAPLE. In [6], it is shown that the BPAS implementation provides exceptional performance, surpassing that of the PowerSeries package, the basic MAPLE function mtaylor, and the multivariate power series available in SageMath [16] by multiple orders of magnitude.

A key design element of our library, in addition to lazy evaluation techniques, is the use of MAPLE objects and object-oriented programming. An object in MAPLE is a special kind of module which encapsulates together data and procedures manipulating that data, just like objects in any other object-oriented language; see [5, Chapters 8, 9]. To the best of our knowledge, few MAPLE libraries make use of those objects, which, as our report suggests, are worth considering for improving performance. In particular, objects allow for the overloading of existing builtin MAPLE functions in order to integrate these new custom objects with existing MAPLE library code. Our results show that MultivariatePowerSeries is comparable in performance to the implementation of BPAS, is thus similarly several orders of magnitude faster than other existing implementations. These experimental results are discussed in Section 6.

The remainder of this paper is organized as follows. We begin in Section 2 with reviewing definitions of formal power series, and univariate polynomial over power series, followed by a brief discussion about the basic arithmetic, Weierstrass preparation theorem and factorization via Hensel’s lemma. Section 3 presents an overview of the MultivariatePowerSeries package, while Section 4 explores its underlying design principles. Implementation details are discussed in Section 5 followed by our experimentation in Section 6. Finally, we conclude and present future works in Section 7.
2 Background

In this section we review the basic properties of formal power series and univariate polynomials over those series, following G. Fischer in [5]. While various proofs of Theorems 1 of 2 can be found in the literature, the proofs given in [6] are constructive and support our implementation. Throughout this paper, \( \mathbb{N} \) denotes the semi-ring of non-negative integers and \( \mathbb{K} \) an algebraic number field.

2.1 Power Series

Given a positive integer \( n \), we denote by \( \mathbb{K}[X_1, \ldots, X_n] \) the set of multivariate formal power series with coefficients in \( \mathbb{K} \) and variables \( X_1, \ldots, X_n \). Let \( f = \sum_{e \in \mathbb{N}^n} a_e X^e \in \mathbb{K}[X_1, \ldots, X_n] \) and \( d \in \mathbb{N} \) where \( X^e = X_1^{e_1} \cdots X_n^{e_n} \) and \( e = (e_1, \ldots, e_n) \in \mathbb{N}^n \). The homogeneous part and polynomial part of \( f \) in degree \( d \) are respectively defined by 
\[
f_d := \sum_{|e| = d} a_e X^e \quad \text{and} \quad f^{(d)} := \sum_{k \leq d} f_k[k],
\]
where \(|e| = e_1 + \cdots + e_n\). The sum (resp. difference) of two formal power series \( f, g \in \mathbb{K}[X_1, \ldots, X_n] \) is defined by the sum (and resp. difference) of their homogeneous parts of the same degree; thus we have: \( f \pm g = \sum_{d \in \mathbb{N}} f_d \pm g_d \). The product \( h = f \cdot g \) can be defined as \( h = \sum_{d \in \mathbb{N}} h_d \) with \( h_d = \sum_{k+l=d} f_k g_l \). With the above addition and multiplication, the set \( \mathbb{K}[X_1, \ldots, X_n] \) is defined by the sum (and resp. difference) of their homogeneous parts of the same degree; thus we have: \( f \pm g = \sum_{d \in \mathbb{N}} f_d \pm g_d \). The product \( h = f \cdot g \) can be defined as \( h = \sum_{d \in \mathbb{N}} h_d \) with \( h_d = \sum_{k+l=d} f_k g_l \). With the above addition and multiplication, the set \( \mathbb{K}[X_1, \ldots, X_n] \) is defined by the sum (and resp. difference) of their homogeneous parts of the same degree; thus we have: \( f \pm g = \sum_{d \in \mathbb{N}} f_d \pm g_d \). The product \( h = f \cdot g \) can be defined as \( h = \sum_{d \in \mathbb{N}} h_d \) with \( h_d = \sum_{k+l=d} f_k g_l \).

The order of the power series \( f \), denoted by \( \operatorname{ord}(f) \), is defined as \( \min \{ d \in \mathbb{N} \mid f_d \neq 0 \} \) if \( f \neq 0 \), and as \( \infty \) otherwise. We observe that \( \mathcal{M}^k = \{ f \in \mathbb{K}[X_1, \ldots, X_n] \mid \operatorname{ord}(f) \geq k \} \) holds for every \( k \geq 1 \). If \( f \) is a unit, that is, if \( f \notin \mathcal{M} \) (or equivalently, if \( \operatorname{ord}(f) = 0 \)) then the sequence \( (h_m)_{m \in \mathbb{N}} \), where \( h_m = c^{-1}(1 + g + \cdots + g^m) \), \( c = f(0) \), and \( g = 1 - c^{-1}f \), converges to the inverse of \( f \). This convergence is the sense of Krull topology, see [5] for details.

2.2 Univariate Polynomials over Power Series

We denote by \( \mathbb{A} \) and \( \mathcal{M} \) the power series ring \( \mathbb{K}[X_1, \ldots, X_n] \) and its maximal ideal. We allow \( n = 0 \), in which case we have \( \mathcal{M} = \langle 0 \rangle \). Let \( f \in \mathbb{A}[X_{n+1}] \), written as \( f = \sum_{i=0}^{\infty} a_i X_{n+1}^i \) with \( a_i \in \mathbb{A} \) for all \( i \in \mathbb{N} \). Then, Weierstrass Preparation Theorem (WPT) states the following.

**Theorem 1.** Assume \( f \neq 0 \mod \mathcal{M}[X_{n+1}] \). Let \( d \geq 0 \) be the smallest integer such that \( a_d \notin \mathcal{M} \). Then, there exists a unique pair \( (\alpha, p) \) satisfying the following:

1. \( \alpha \) is an invertible power series of \( \mathbb{A}[X_{n+1}] \),
2. \( p \in \mathbb{A}[X_{n+1}] \) is a monic polynomial of degree \( d \),
3. writing \( p = X_{n+1}^d + b_{d-1}X_{n+1}^{d-1} + \cdots + b_1 X_{n+1} + b_0 \), we have \( b_{d-1}, \ldots, b_0 \in \mathcal{M} \),
4. \( f = \alpha p \) holds.

Moreover, if \( f \) is a polynomial of \( \mathbb{A}[X_{n+1}] \) of degree \( d + m \), for some \( m \), then \( \alpha \) is a polynomial of \( \mathbb{A}[X_{n+1}] \) of degree \( m \).
Since $A$ is a UFD, then Gauss’ lemma implies that the polynomial ring $A[X_{n+1}]$ is also a UFD. Hensel’s lemma shows how factorizing a polynomial in $A[X_{n+1}]$ can be reduced to factorizing a polynomial in $K[X_{n+1}]$.

**Theorem 2 (Hensel’s Lemma).** Assume that $f$ is a polynomial of degree $k$ in $A[X_{n+1}]$. We define $\overline{f} = f(0, \ldots, 0, X_{n+1}) \in K[X_{n+1}]$. We assume that $f$ is monic in $X_{n+1}$, that is, $a_k = 1$. We further assume that $K$ is algebraically closed. Thus, there exists positive integers $k_1, \ldots, k_r$ and pairwise distinct elements $c_1, \ldots, c_r \in K$ such that we have $\overline{f} = (X_{n+1} - c_1)^{k_1}(X_{n+1} - c_2)^{k_2} \cdots (X_{n+1} - c_r)^{k_r}$.

Then, there exists $f_1, \ldots, f_r \in A[X_{n+1}]$, all monic in $X_{n+1}$, such that we have:

1. $f = f_1 \cdots f_r$,
2. the degree of $f_j$ is $k_j$, for all $j = 1, \ldots, r$,
3. $\overline{f_j} = (X_{n+1} - c_j)^{k_j}$, for all $j = 1, \ldots, r$.

3 An Overview of the User-Interface

From the point of view of the end-user, the **MultivariatePowerSeries** package is a collection of commands for manipulating multivariate power series and univariate polynomials over multivariate power series. The field of coefficients of all power series created by the command **PowerSeries** consists of all complex numbers that are constructible in MAPLE, thus including rational numbers and algebraic numbers. The main algebraic functionalities of this package deal with arithmetic operations (addition, multiplication, inversion, evaluation), for both multivariate power series and univariate polynomials over multivariate power series (UPoPS), as well as factorization of such polynomials. The list of the exposed commands is given in Figure 1.

> with(MultivariatePowerSeries);

```
[Add, ApproximatelyEqual, ApproximatelyZero, Copy, Degree, Display, Divide,
 EvaluateAtOrigin, Exponentiate, GeometricSeries, GetAnalyticExpression,
 GetCoefficient, HenselFactorize, HomogeneousPart, Inverse, IsUnit,
 MainVariable, Multiply, Negate, PowerSeries, Precision,
 SetDefaultDisplayStyle, SetDisplayStyle, Subtract, SumOfAllMonomials,
 TaylorShift, Truncate, UnivariatePolynomialOverPowerSeries,
 UpdatePrecision, Variables, WeierstrassPreparation]
```

Fig. 1: List of the commands of MultivariatePowerSeries.

The commands **PowerSeries** and **UnivariatePolynomialOverPowerSeries** create power series and univariate polynomials over multivariate power series, respectively, from objects like polynomials, sequences, and functions which produce homogeneous parts of a power series, as illustrated in Figures 2 and 3. The commands **GeometricSeries** and **SumOfAllMonomials** respectively create the geometric series and sum of all monomials for an input list of variables.
Multivariate Power Series in Maple

\[ a := \text{PowerSeries}(1 + x + x y + x^2); \]
\[ b := \frac{1}{3}; \]
\[ \text{Truncate}(b, 5); \]
\[ c := \text{PowerSeries}\left( \frac{2x^3}{31}, \text{analytic} = \text{exp}(a) \right); \]
\[ \text{Truncate}(c, 5); \]
\[ \text{Truncate}(c, 10); \]
\[ \frac{1}{1 + x + \frac{1}{2} x^2 + \frac{1}{6} x^3 + \frac{1}{24} x^4 + \frac{1}{120} x^5 + \frac{1}{720} x^6 + \frac{1}{5040} x^7 + \frac{1}{40320} x^8 + \frac{1}{362880} x^9 + \frac{1}{362880} x^{10} \]

Fig. 2: Creating power series from a polynomial or an anonymous function.

\[ a := \text{GeometricSeries}([x, y]); \]
\[ \text{GetAnalyticExpression}(a); \]
\[ b := \text{PowerSeries}(3 + 2 x + y); \]
\[ c := \text{PowerSeries}\left( \frac{x^3}{6}, \text{analytic} = \text{exp}(a) \right); \]
\[ \text{GetAnalyticExpression}(c); \]
\[ \frac{1}{1 - x - y} + \frac{1}{3 + 2 x + y} + \frac{1}{6} x^3 + \frac{1}{24} x^4 + \frac{1}{120} x^5 + \frac{1}{720} x^6 + \frac{1}{5040} x^7 + \frac{1}{40320} x^8 + \frac{1}{362880} x^9 \]

Fig. 3: Creating a univariate polynomial over power series.

\[ a := \text{GeometricSeries}([x, y]) + \text{SumOfAllNonomials}([x, y]); \]
\[ \text{Display}(a); \]
\[ \text{Truncate}(a, 10); \]
\[ \text{Display}(a, \text{maxterms} = 20, \text{precision} = 5); \]
\[ \text{Display}(a, \text{precision} = 5); \]
\[ \frac{1}{1 - x - y} + \frac{1}{(1 - x)(1 - y)} + \frac{2 + 2 x + 2 y}{2} + \frac{2 + 2 x + 2 y}{2} \]

Fig. 4: Controlling the output format of a multivariate power series.
Whenever possible, the package associates every power series with its so-called analytic expression. For each power series \( s \), created by the command `PowerSeries` as the image of a polynomial \( p \) (under the natural embedding from \( \mathbb{C}[X_1, \ldots, X_n] \) to \( \mathbb{C}[[X_1, \ldots, X_n]] \)) the polynomial \( p \) is the analytic expression of \( s \). If a power series is defined by the sequence of its homogeneous parts, as illustrated on Figure 3, the user can optionally specify the sum of that series which is then set to its analytic expression. Power series that have an analytic expression are closed under addition, multiplication and inversion. Propagating that information provides the opportunity to speed up some computations and make decisions that could not be made otherwise. For instance, the command `HenselFactorize` needs to decide whether its input polynomial has an invertible leading coefficient; to do it starts by checking whether the analytic expression of that leading coefficient is known and equal to one.

The commands `Display`, `SetDefaultDisplayStyle` and `SetDisplayStyle` control the output format of multivariate power series and UPoPS. Meanwhile, the commands `HomogeneousPart`, `Truncate`, `GetCoefficient`, `Precision`, `Degree`, `MainVariable` access data from a power series or a univariate polynomial over power series, as illustrated by Figure 4.

The commands `Add`, `Negate`, `Multiply`, `Exponentiate`, `Inverse`, `Divide`, `EvaluateAtOrigin`, and `TaylorShift` perform arithmetic operations on multivariate power series and univariate polynomials over multivariate power series. The functionality of the first six commands can also be accessed using the standard arithmetic operators. As will be discussed in Sections 4 and 5, the implementation of every arithmetic operation, such as addition, multiplication, inversion builds the resulting power series (sum, product or inverse) “lazily”, by creating its generator from the generators of the operands, which are called ancestors of the resulting power series.

```
> f := UnivariatePolynomialOverPowerSeries(PowerSeries(x), GeometricSeries(y), PowerSeries(1),
      PowerSeries([1 + x + y], [x]));
> f := [UnivariatePolynomialOverPowerSeries: (x)+(1+y+...)z+(1+...)z^2]
> p, a := WeierstrassPreparation(f);
> p := [UnivariatePolynomialOverPowerSeries: (1)], [UnivariatePolynomialOverPowerSeries: 
       (-6)+([1+ x] z+(−6) + x) z^2+(1) z^3]
> UpdatePrecision(p, 5));
> a := [UnivariatePolynomialOverPowerSeries: (x+x^2−xy+y^2−x^2y+y^2−x^2y+y^2+...) 
       +([1+ x] z+(−6) + x) z^2+(1) z^3]
> h := p\*a;
> h := [UnivariatePolynomialOverPowerSeries: (x+...)+(1+y+y^2+y^3+...)z+(1+...)z^2+(1−x+y^2+2xy+y^3+...)z^3]
> ApproximatelyEqual(f, h, 20);
```

Fig. 5: Factoring univariate polynomials using `WeierstrassPreparation`. 

\[ \text{Fig. 5: Factoring univariate polynomials using } \text{WeierstrassPreparation}. \]
Multivariate Power Series in Maple

> \( f := \text{UnivariatePolynomialOverPowerSeries}\left( (z-1)\cdot(z-2)\cdot(z-3) + x\cdot(z^2 + z), \, z \right); \)
\[ f = \left[ \text{UnivariatePolynomialOverPowerSeries}((-6) + (11 + x) z + (-6 + x) z^2 + (1 + 1) z^3) \right] \] (22)

> \( F := \text{HenselFactorize}\left( f \right); \)
\[
F = \left[ \text{UnivariatePolynomialOverPowerSeries}((-1 + \ldots) + (1) z), \text{UnivariatePolynomialOverPowerSeries}(-2 + \ldots + (1) z) \right] \left[ \text{UnivariatePolynomialOverPowerSeries}(-3 + \ldots + (1) z) \right] \] (23)

> \text{map}(\text{UpdatePrecision}, F, 5);

\[
\left[ \text{UnivariatePolynomialOverPowerSeries}(-1 + x - 3 x^2 + \frac{27 x^3}{2} - \frac{291 x^4}{8} + \ldots + (1) z), \right.
\]
\[
\left. \text{UnivariatePolynomialOverPowerSeries}(-2 - 6 x - 30 x^2 - 602 x^3 - 5610 x^4 - 93390 x^5 + \ldots + (1) z), \right. \]
\[
\left. \text{UnivariatePolynomialOverPowerSeries}(-3 + 6 x + 33 x^2 + \frac{777 x^3}{2} + \frac{2273 x^4}{4} + \frac{74565 x^5}{8} + \ldots + (1) z) \right] \] (24)

> \( h := f[1]:f[2]:f[3] - f \)
\[ h = \left[ \text{UnivariatePolynomialOverPowerSeries}((0 + \ldots) + (0 + \ldots) z + (0 + \ldots) z^2 + (0) z^2) \right] \] (25)

> \text{ApproximatelyEqual}(h, 100); \text{true} \]

> \( g := \text{UnivariatePolynomialOverPowerSeries}\left( y^2 + x^2 + (y + 1) \cdot z^2 + z^3, \, z \right); \)
\[ g = \left[ \text{UnivariatePolynomialOverPowerSeries}((x^2 + y^2 + (0 + \ldots) z + (1 + y) z^2) + (1 + \ldots + (1) z) \right] \] (27)

> \( G := \text{HenselFactorize}(g); \)
\[
G = \left[ \text{UnivariatePolynomialOverPowerSeries}((0 + \ldots) + (0 + \ldots) z + (1) z^2), \text{UnivariatePolynomialOverPowerSeries}(1 + \ldots + (1) z) \right] \] (28)

> \text{map}(\text{UpdatePrecision}, G, 8);

\[
\left[ \text{UnivariatePolynomialOverPowerSeries}(-6 y^2 - 3 x^2 y - 43 x^2 y^2 - 24 x^2 y^3 - 12 x^2 y^4 + 36 x^2 y^5 + 145 x^2 y^6 + 38 x^2 y^7 + \ldots + (-y - y^2 + 2 y^3 + y^4 + 2 x y^2 + 2 x^2 y^3 + 10 x y^3 + 16 x^2 y^4 + 6 y^4 + 7 y^5 + 9 x^2 y^5 + 34 x y^5 + 18 y^5 + 56 X y^6 + 98 x^2 y^6 + 34 x y^6 + 8 y^6 + 30 x^2 y^7 + 132 x^2 y^8 + 436 x^2 y^9 + 363 x^3 y^9 + 89 y^9 + \ldots + (1) z) + x + y + z + y^2 - 2 y^2 + 2 x y - 2 x^2 y^2 + 10 x y^2 + 16 x^2 y^3 + 6 y^3 + 7 x^2 y^3 + 9 x y^3 + 34 x y^3 + 18 y^3 + 56 x y^4 + 98 x^2 y^4 + 34 x y^4 + 8 y^4 + 30 x^2 y^5 + 132 x^2 y^6 + 436 x^2 y^7 + 363 x^3 y^7 + 89 y^7 + \ldots + (1) z) \right] \left[ \text{UnivariatePolynomialOverPowerSeries}(1 + y + x^2 + 8 y^2 - 2 x^2 y^2 - 2 x^2 y^3 + y + 10 x y^2 + 16 x^2 y^3 + 6 y^3 + 7 x^2 y^3 + 9 x y^3 + 34 x y^3 + 18 y^3 + 56 x y^4 + 98 x^2 y^4 + 34 x y^4 + 8 y^4 + 30 x^2 y^5 + 132 x^2 y^6 + 436 x^2 y^7 + 363 x^3 y^7 + 89 y^7 + \ldots + (1) z) \right] \] (29)

> \( h := g[1]:g[2] - g \)
\[ h = \left[ \text{UnivariatePolynomialOverPowerSeries}((x^2 + y^2 + \ldots) + (0 + \ldots) z + (1 + y + \ldots) z^2 + (1) z^2) \right] \] (30)

> \text{ApproximatelyEqual}(g, h, 20); \text{true} \]

Fig. 6: Factoring univariate polynomials using \texttt{HenselFactorize}.

The commands \texttt{WeierstrassPreparation} and \texttt{HenselFactorize} factorize univariate polynomials over multivariate power series. Thanks to their implementation based on lazy evaluation, each of these factorization commands returns the factors as soon as enough information is discovered for initializing the data structures of the factors; see Figures 5 and 6.

The precision of each returned factor, that is, the common precision of its coefficients (which are power series) is zero. However, the generator (see Section 4) for this term of each coefficient is known and, thus, the computation of more coefficients can be resumed when a higher precision is requested. Such a request can be explicit by calling \texttt{UpdatePrecision}, or implicit, when requesting data of a higher precision than has been previously requested through, e.g., \texttt{Truncate} or \texttt{HomogeneousPart}.
4 Design Principles

In this section we examine several design principles underpinning the implementation of the MultivariatePowerSeries library. Foremost is lazy evaluation: an algorithmic technique where the computation of data is postponed until explicitly required (Section 4.1). The eventual implementations of these lazy-evaluation algorithms make deliberate efforts to use appropriate MAPLE data structures and built-in functions to optimize performance (Section 4.2). Lastly, in support of software quality and integration with existing MAPLE library code, we employ MAPLE’s object-oriented mechanisms (Section 4.3).

4.1 Lazy Evaluation

Lazy evaluation is an optimization technique most commonly appearing in the study of functional programming languages [9]. The lazy evaluation or “call-by-need” refers to delaying the call to a function until its result is genuinely needed. This is often complemented by storing the result for later look-up.

In the case of power series, consider a bivariate geometric series $f = \sum_{d=0}^{\infty} f(d)$ where $f(0) = 1, f(1) = x + y, f(2) = x^2 + 2xy + y^2, \ldots, f(d) = (x + y)^d$. One can prove that $f$ converges to $\frac{1}{1-x-y}$. Of course, in practice, it is impossible to store an infinite number of terms on a computer with finite memory. A naïve implementation then suggests storing $f(d)$ for some large and predetermined $d$. Thus, one can approximate power series as multivariate polynomials. Such an implementation could be called truncated power series.

While this representation of power series is easy to implement, it leads to notable restrictions for the study of formal power series. First, one must a priori determine the precision, i.e. the particular value of $d$. Second, in a most naïve implementation, previously-computed homogeneous parts must be recomputed whenever a new, greater precision is required. For example, the polynomial $f^{(d+1)}$ is likely to be constructed “from scratch” despite the polynomial $f^{(d)}$ possibly being already computed. Third, storing and manipulating the polynomial part of a power series up to a degree $d$ needs a large portion of memory. This latter problem is exacerbated when the predetermined precision is not a tight upper bound on the required precision.

To combat the challenges of a truncated power series implementation, we take advantage of lazy evaluation. Every power series is represented by a unique procedure to compute a homogeneous part for a given degree. For example, Listing 1.1 shows such a procedure for the bivariate geometric series which converges to $\frac{1}{1-x-y}$. As we will see, this lazy evaluation design can be paired with an array of polynomials storing the previously computed homogeneous parts.

```
1 generator := proc (d :: nonnegint)
2     return expand((x+y)^d);
3 end proc;
```

Listing 1.1: A MAPLE implementation of $f(d)$ in $\frac{1}{1-x-y} = \sum_{d=0}^{\infty} f(d)$. 

4.2 Maple Data Structures and Built-in Functions

Using an appropriate data structure for encoding and manipulating data is critical for performance, particularly in high-level and interpreted programming languages like MAPLE. In MAPLE, modifying an existing list or set—such as by appending, replacing, or deleting an element—leads to the creation of a new list or set, rather than modifying the original one in-place. In contrast, an Array is a low-level and mutable data-structure which allows for in-place modification of its elements. These functionalities provide much better performance than lists or sets when the collection is frequently changed or when the elements being modified are themselves large in size. This fact is clear from the overwhelming improvement in performance of our library compared against the existing PowerSeries library which uses lists to encode homogeneous parts; see Section 6.

Looking more closely at the Array data structure, an \( n \)-dimensional Array is stored as a \( n \)-dimensional rectangular block named RTABLE. The length of the associated RTABLE is \( 2^n + d \) where \( d \) is maximum number of elements that may be stored, i.e., the allocation size of the Array; see [5, Appendix 1]. For the storage of homogeneous parts of a power series, and the power series coefficients of a UPoPS, we utilize 1-dimensional Arrays. Listing 1.2 in the next section shows this as the variables hpoly and upoly, respectively.

To further improve performance, we make use of low-level built-in functions. Such functions are provided as compiled code within the MAPLE kernel, and therefore not written in the MAPLE language. Most notably, instead of using MAPLE for-loops and the typical + and * syntaxes for addition and multiplication, respectively, we reduce the cost of summations and multiplications remarkably by taking advantage of built-in MAPLE functions, add and mul. These built-in functions, respectively, add or multiply the terms of an entire sequence of expressions together to return a single sum or product. These functions avoid a large number of high-level function calls and reduce memory usage by avoiding copying and re-allocation of data.

4.3 Maple Objects

An often overlooked aspect of MAPLE is its object-oriented capability. An object allows for variables and procedures operating on that data to be encapsulated together in a single entity. In MAPLE, a class—the definition of a particular type of object—can be declared by including the option object in a module declaration. Evaluating this declaration returns an object of that class. This new object is often a so-called “prototype” object which, when passed to the Object routine, returns a new object of the same class. See [5, Chapter 9] for further details on object-oriented programming in MAPLE.

Our power series and UPoPS types are implemented using these object-oriented features of MAPLE. The classes for each are named, respectively, PowerSeriesObject and UnivariatePolynomialOverPowerSeriesObject.
The use of object-oriented programming in MAPLE has two key benefits: (i) the organization object-oriented code provides better software quality through modularity and maintainability; and (ii) allows for the overloading of built-in functions, thus allowing objects to be integrated with, and used natively by, existing MAPLE library functions.

```plaintext
MultivariatePowerSeries := module ()
  option package;
  local PowerSeriesObject, UnivariatePolynomialOverPowerSeriesObject;
  # create a power series:
  export PowerSeries := proc (...)
    # create a UPoPS:
    export UnivariatePolynomialOverPowerSeries := proc (...)
      # Additional procedures to interface these two classes

      module PowerSeriesObject ()
        option object;
        local hpoly :: Array, precision :: nonnegint, generator :: procedure;
        # other members and methods
        end module;

      module UnivariatePolynomialOverPowerSeriesObject ()
        option object;
        local upoly :: Array, vname :: name;
        # other members and methods
        end module;

Listing 1.2: An overview of the MultivariatePowerSeries package.
```

The MultivariatePowerSeries library contains a package of the same name which groups together those two aforementioned classes along with additional procedures to construct and manipulate objects of those classes. These additional procedures are used to “hide” the object-oriented nature of the library behind simple procedure calls. This keeps the package syntactically and semantically consistent with the general paradigm of MAPLE which does not use object-oriented programming. As an example of such a procedure, PowerSeries, as seen in Fig. 2 (Section 3), handles various different types of input parameters to correctly construct a PowerSeriesObject object through delegation to the correct class method.

Listing 1.2 shows the declaration of our two classes and the MultivariatePowerSeries package. The latter is created by using option package in a module declaration; see [5, Chapter 8]. The implementation of these two classes is further discussed in Section 5.
5 Implementation of MultivariatePowerSeries

The MultivariatePowerSeries package provides a collection of procedures which form simple wrappers for the methods of the aforementioned classes, PowerSeriesObject and UnivariatePolynomialOverPowerSeriesObject. These classes, respectively, define the data structures and algebraic functionalities for creating and manipulating multivariate power series and univariate polynomials over power series. This section discusses those data structures as well as the implementation of basic arithmetic, Weierstrass Preparation Theorem, and factorization via Hensel's lemma, all following a lazy evaluation scheme.

5.1 PowerSeriesObject

The PowerSeriesObject class provides basic arithmetic operations, like addition, multiplication, inversion, and evaluation, for multivariate power series, all utilizing lazy evaluation techniques. Let \( f \in \mathbb{K}[X_1, \ldots, X_n] \) be a non-zero multivariate power series defined as \( f = \sum_{d=0}^{\infty} f(d) \). \( f \) is encoded as an object of type PowerSeriesObject, containing the following attributes.

First, the power series generator is the procedure to compute \( f(d) \), the \( d \)-th homogeneous part of \( f \), for \( d \in \mathbb{N} \). Second, the precision is a non-negative integer encoding the maximum degree of the homogeneous parts which have so far been computed. Third, the 1-dimensional array storing the previously computed homogeneous parts of \( f \), denoted as \( \text{hpoly} \) in Listing 1.2.

To create a power series object this class provides a variety of constructors. Power series objects may be created from polynomials, algebraic numbers, UP-oPS objects, or procedures defining the generator of the power series.

Every arithmetic operation returns a lazily-constructed power series object by creating its generator from the generators of the operands, but without explicitly computing any homogeneous parts of the result. Thus, this is a lazy power series, so that the homogeneous parts of the result are computed when truly needed. Once homogeneous parts are eventually computed, they are stored in the array \( \text{hpoly} \). An important aspect of this organization is that the generator of the resulting power series becomes implicitly connected to the generators of the operands; the latter are thus called the ancestors of the former. Note that the ancestors are merely stored as references, not copies, thus saving time and memory resources.

Moreover, the addition and multiplication operations are not only binary operations (operations taking two parameters), but are \( m \)-ary operations. For multiplication, a sequence of power series \( f_1, \ldots, f_m \in \mathbb{K}[X_1, \ldots, X_n] \) may be passed to the multiplication algorithm to produce the product \( f_1 \cdot f_2 \cdots f_m \) via lazy evaluation. Similarly, addition may take the sequence \( f_1, \ldots, f_m \) to return the sum \( f_1 + f_2 + \cdots + f_m \). Further, addition may also take as a parameter an optional sequence of polynomial coefficients \( c_1, \ldots, c_m \in \mathbb{K}[X_1, \ldots, X_n] \) to return the sum \( c_1f_1 + \cdots + c_m f_m \) constructed lazily.

A key part to the efficiency of lazy evaluation is to not re-compute any data. We have already seen that the \( \text{hpoly} \) array stores previously computed
homogeneous parts for a PowerSeriesObject object. What is missing is to ensure that the array is accessed where possible rather than calling the generator function. Moreover, one must avoid directly accessing that array for homogeneous parts which are not yet computed. We thus provide the function HomogeneousPart(f, d), demonstrated in Listing 1.3 to handle both of these cases. This function returns the d-th homogeneous part of the power series f; if d is greater than the precision (f:-precision), then this method iteratively calls the generator to update hpoly and precision, otherwise it simply returns the previously computed homogeneous part. From here on we use hpart as shorthand for the HomogeneousPart function.

Listing 1.3: A simplified version of the HomogeneousPart function in PowerSeriesObject.

Listing 1.4 shows a simplified implementation of Divide that computes the quotient of two power series objects \( f, g \in K[X_1, \ldots, X_n] \). In particular, notice the creation of the local procedure gen for the generator of the quotient. Note that EXPAND is a local macro defined in MultivariatePowerSeries to efficiently perform expansion and normalization supporting algebraic inputs.

Listing 1.4: A simplified version of the division method in PowerSeriesObject.

5.2 UnivariatePolynomialOverPowerSeriesObject
The UnivariatePolynomialOverPowerSeriesObject class is implemented as a simple dense univariate polynomial with the simple and obvious implementations
of associated arithmetic (see, e.g., [18] Chapter 2). The arithmetic operations are achieved directly from coefficient arithmetic, that is, \texttt{PowerSeriesObject} arithmetic. Since the latter is implemented using lazy evaluation techniques, UPoPS arithmetic is inherently and automatically lazy.

For example, the addition of two UPoPS objects $f = \sum_{i=0}^{k} a_i x_{n+1}^i$ and $g = \sum_{i=0}^{k} b_i x_{n+1}^i$ in $\mathbb{K}[X_1, \ldots, X_n][X_{n+1}]$ is the summation $(a_i + b_i) x_{n+1}^i$ for all $0 \leq i \leq k$, where $a_i, b_i$ are \texttt{PowerSeriesObject} objects. Other basic arithmetic operations behave similarly. However, there are important operations on UPoPS which are not as straightforward. In the following we explain our implementation of Weierstrass Preparation Theorem, Taylor shift, and factorization via Hensel’s lemma for UPoPS, all of which follow lazy evaluation techniques.

**Weierstrass Preparation.** Let $f, p, \alpha \in \mathbb{K}[X_1, \ldots, X_n][X_{n+1}]$ be such that they satisfy the conditions of Theorem 1 and such that $f = \sum_{i=0}^{d+m} a_i x_{n+1}^i$, $p = X_{n+1}^d + \sum_{i=0}^{d-1} b_i x_{n+1}^i$, and $\alpha = \sum_{i=0}^{m} c_i x_{n+1}^i$. Equating coefficients in $f = p\alpha$ we derive the two following systems of equations:

$$
\begin{align}
\begin{cases}
\quad a_0 &= b_0 c_0 \\
\quad a_1 &= b_0 c_1 + b_1 c_0 \\
\quad & \vdots \\
\quad a_{d-1} &= b_0 c_{d-1} + b_1 c_{d-2} + \cdots + b_{d-2} c_1 + b_{d-1} c_0 \\
\quad a_d &= b_0 c_d + b_1 c_{d-1} + \cdots + b_{d-1} c_1 + c_0 \\
\quad & \vdots \\
\quad a_{d+m-1} &= b_{d-1} c_m + c_{m-1} \\
\quad a_{d+m} &= c_m
\end{cases}
\end{align}
$$

To solve these systems we proceed by solving them modulo successive powers of $\mathcal{M}$, following the proof of Theorem 1 in [4]. Notice that solving modulo successive powers of $\mathcal{M}$ is precisely the same as computing homogeneous parts of increasing degree. Thus, this follows our lazy evaluation scheme perfectly. The power series $b_0, \ldots, b_{d-1}$ are generated by Equations (1) and $c_0, \ldots, c_m$ by Equations (2).

Consider that $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ are known modulo $\mathcal{M}^r$ while $a_0, \ldots, a_{d-1}$ are known modulo $\mathcal{M}^{r+1}$; this latter fact is simple since $f$ is the input to Weierstrass Preparation and is fully known. From the first equation in (1), $b_0$ can be computed modulo $\mathcal{M}^{r+1}$ since $b_0 \in \mathcal{M}$, $c_0$ is known modulo $\mathcal{M}^r$, and $a_0$ is known $\mathcal{M}^{r+1}$. Then, the equation $a_1 = b_0 c_1 + b_1 c_0$, that is, $a_1 - b_0 c_1 = b_1 c_0$ can be solved for $b_1$ modulo $\mathcal{M}^{r+1}$ since, again, $b_1 \in \mathcal{M}$ and the other terms are sufficiently known. We compute all $b_2, \ldots, b_{d-1}$ modulo $\mathcal{M}^{r+1}$ with the same argument. After determining $b_0, \ldots, b_{d-1}$ modulo $\mathcal{M}^{r+1}$, we can compute $c_m, c_{m-1}, \ldots, c_0$ modulo $\mathcal{M}^{r+1}$ from Equations (2) with simple power series multiplication and subtraction, working iteratively, in a bottom up fashion. For example, $c_{m-1} = a_{d+m-1} - b_{d-1} c_m$. 


As yet, we have not explicitly seen how the coefficients of \( p \) and \( \alpha \) will be updated. The key idea is that to update a single power series coefficient of \( p \) or \( \alpha \) requires simultaneously updating all coefficients of \( p \) and \( \alpha \). Thus, all the generators of \( b_0, \ldots, b_{d-1}, c_0, \ldots, c_m \) simply call a single “Weierstrass update” function to update all power series simultaneously using Equations (1) and (2). Algorithm 1 shows this Weierstrass update function.

**Algorithm 1** WeierstrassUpdate\((p, \alpha, F, r)\)

Given \( p = X_{n+1}^d + \sum_{i=0}^{d-1} b_i X_{n+1}^i, \alpha = \sum_{i=0}^m c_i X_{n+1}^i, r \in \mathbb{N}, \) and \( F = \{ F_i \mid F_i = a_i - \sum_{j=0}^{i-1} b_j c_{i-j}, 0 \leq i < d \} \) are all known modulo \( \mathcal{M} \), returns \( b_0, \ldots, b_{d-1}, c_0, \ldots, c_m \) modulo \( \mathcal{M}^{r+1} \).

1. **# update \( b_0, \ldots, b_{d-1} \) modulo \( \mathcal{M}^{r+1} \)**
   1. for \( i \) from 0 to \( d - 1 \) do
   2. \( s := \text{add}\left(\text{seq}(\text{hpart}(b_i, r - k) \cdot \text{hpart}(c_0, k), k = 1 \ldots r - 1)\right); \)
   3. \( \text{hpart}(b_i, r) := (\text{hpart}(F_i, r) - s)/\text{hpart}(c_0, 0); \)

4. **# ensure \( c_0, \ldots, c_m \) are updated modulo \( \mathcal{M}^{r+1} \)**
   1. for \( i \) from 0 to \( m \) do
   2. \( \text{hpart}(c_i, r); \)

In order to update the coefficients of \( p \), we frequently need to compute \( a_i - \sum_{j=0}^{i-1} b_j c_{i-j} \) for \( 0 \leq i < d \). To optimize this operation, we a priori create helper power series as the set \( F = \{ F_i \mid F_i = a_i - \sum_{j=0}^{i-1} b_j c_{i-j}, i = 0, \ldots, d - 1 \} \). The power series \( F_i \), following power series arithmetic with lazy evaluation, allows for the efficient computation of homogeneous parts of increasing degree of \( a_i - \sum_{j=0}^{i-1} b_j c_{i-j} \). This set \( F \) is passed to the Weierstrass update function to optimize the overall computation.

Finally, the Weierstrass preparation must be initialized before continuing with Weierstrass updates. Namely, the degree of \( p \) and the initial values of \( p \) and \( \alpha \) modulo \( \mathcal{M} \) must first be computed. The degree of \( p \), namely \( d \), is set to be the smallest integer \( i \) such that \( a_i \) is a unit. If \( d = 0 \), then \( p = 1 \) and \( \alpha = f \), otherwise, \( m \) equals the difference between the degree of \( f \) and \( d \), and we initialize \( b_i = 0 \) for \( 0 \leq i < d \). Then, \( c_m, \ldots, c_0 \) are initialized using power series arithmetic following Equations (2). Lastly, the set \( F \) is initialized.

**Taylor Shift.** This operation takes a UPoPS object \( f \in \mathbb{K}[X_1, \ldots, X_n][X_{n+1}] \) and performs the translation \( X_{n+1} \rightarrow X_{n+1} + c \), i.e. \( f(X_{n+1} + c) \), for some \( c \in \mathbb{K} \). In our implementation, \( c \) can be a numeric or algebraic MAPLE type with the purpose of being used efficiently in factorization via Hensel’s Lemma.

Assume \( f = \sum_{i=0}^{k} a_i X_{n+1}^i \) is a UPoPS in \( \mathbb{K}[X_1, \ldots, X_n][X_{n+1}] \) and \( c \in \mathbb{K} \). As the PowerSeriesObject objects \( a_0, \ldots, a_k \) are lazily evaluated power series, we want to also make Taylor shift a lazy operation. Thus, we need to create a generator for the power series coefficients of \( f(X_{n+1} + c) \). Let \( \mathcal{T} = (t_{i,j}) \) be the lower triangular matrix of the coefficients of \( X_{n+1}^i \) in the binomial expansion \( (X_{n+1} + c)^i \), for \( 0 \leq i \leq k \), and \( 0 \leq j \leq i \). Let \( (b_0, \ldots, b_k) \) be the list of coefficients of \( f(X_{n+1} + c) \) in \( \mathbb{K}[X_1, \ldots, X_n] \). Then, it is easy to prove that for every \( 0 \leq i \leq k \), \( b_i \) is the inner product of the \( i \)-th sub-diagonal of \( \mathcal{T} \) with the lower
$k+1-i$ elements of the vector $(a_0, \ldots, a_k)$. This inner product can be computed efficiently by taking advantage of the $m$-ary addition operation described for the PowerSeriesObject (see Section 5.1). Since this operation returns a lazily-constructed power series, this precisely defines the lazy construction of the power series $b_0, \ldots, b_k$, thus making Taylor shift a lazy operation.

**Factorization via Hensel’s Lemma.** Hensel’s lemma for factorizing univariate polynomials over power series was reviewed in Theorem 2, where $K$ is algebraically closed and $f \in K[[X_1, \ldots, X_n]][X_{n+1}]$ is a UPoPS object. Following the ideas of [6], we compute the factors of $f$ in a lazy fashion. Algorithm 2 proceeds through iterative applications of Taylor shift and Weierstrass Preparation Theorem in order to create one factor of $f$ at a time. Those factors are actually computed through lazy evaluation thanks to the lazy behavior of the procedures WeierstrassPreparation and TaylorShift. This Algorithms thus computes and updates the factors modulo the successive powers $M, M^2, M^3, \ldots$ of the maximal ideal $M$.

---

**Algorithm 2** HenselFactorize($f$)  
Given $f = \sum_{i=0}^{k} a_i X_{n+1}^i \in K[X_1, \ldots, X_n][X_{n+1}]$, returns a list of factors $\{f_1, \ldots, f_r\}$ so that $f = a_k \cdot f_1 \cdots f_r$, and satisfies Theorem 2.

1: if $a_k \notin M$ then  
2: $f^* := \frac{1}{a_k} \cdot f$;  
3: else  
4: error “$a_k$ must be a unit.”  
5: $\bar{f} := \text{EvaluateAtOrigin}(f)$;  
6: $c_1, \ldots, c_r := \text{Roots}(\bar{f}, X_{n+1})$;  
7: for $i$ from 1 to $r$ do  
8: $g := \text{TaylorShift}(f^*, c_i)$;  
9: $p, \alpha := \text{WeierstrassPreparation}(g)$;  
10: $f_i := \text{TaylorShift}(p, -c_i)$;  
11: $f^* := \text{TaylorShift}(\alpha, -c_i)$;  
12: return $\{f_1, \ldots, f_r\}$;

---

Note that the generation of the factors $f_1, \ldots, f_r$ takes place after factorizing $\bar{f} \in K[X_{n+1}]$. Recall that $\bar{f}$ is obtained by evaluating each $X_i$ to 0 for $1 \leq i \leq n$. This is called EvaluateAtOrigin in our implementation. To efficiently factor $\bar{f}$, we take advantage of the package SolveTools [13], which allows us to compute the splitting field of $\bar{f}$ (which, in practice, is a polynomial with coefficients in some algebraic extension of $\mathbb{Q}$) and factorize $\bar{f}$ into linear factors.

Let $c_1, \ldots, c_r$ be the distinct roots of $\bar{f}$ and $k_1, \ldots, k_r$ their respective multiplicities. To describe one iteration of Algorithm 2 let $f^*$ be the current polynomial to factorize. For a root $c_i$ of $\bar{f}$, and thus $f^*$, we perform a Taylor shift to obtain $g = f^*(X_{n+1} + c_i)$. Then, we apply Weierstrass preparation on $g$ to obtain $p$ and $\alpha$ where $p$ is monic and of degree $k_i$. Again, by using Taylor Shift, we apply the reverse shift to $p$ to obtain $f_i = p(X_{n+1} - c_i)$, a factor of $f$, and
\[ f^* = \alpha(X_{n+1} - c_i), \]

for the next iteration. As mentioned above, since both Taylor shift and Weierstrass preparation are implemented using lazy evaluation, our factorization via Hensel’s lemma is inherently lazy.

6 Experimentation

We compare the performance of the \texttt{MultivariatePowerSeries} package, denoted MPS, with the previous MAPLE implementation of multivariate power series, the \texttt{PowerSeries} package, denoted RCPS, and the recent implementation of power series via lazy evaluation in the BPAS library. This latter implementation is written in the C language on top of efficient sparse multivariate arithmetic; see [4,6]. It has already been shown in [6] that the implementation in BPAS is orders of magnitude faster than the \texttt{PowerSeries} package, MAPLE’s \texttt{mtaylor} command, and the multivariate power series available in SAGEMATH. As we will see, our implementation performs comparably to that of BPAS.

Throughout this section, we collect our benchmarks on a machine running Ubuntu 18.04.4, MAPLE 2020, and BPAS (ver. 1.652), with an Intel Xeon X5650 processor running at 2.67GHz, with 12x4GB DDR3 memory at 1.33 GHz.

Figures 7 and 8 respectively, show the performance of division and multiplication algorithms to compute \( \frac{1}{f} \) and \( \frac{1}{f} \cdot f \) for power series \( f_1 = 1 + X_1 + X_2 \), \( f_2 = 1 + X_1 + X_2 + X_3 \), and \( f_3 = 2 + \frac{1}{3}(X_1 + X_2) \). It can be seen that MPS power series division is 9×, 2100×, and 3× faster than the previous MAPLE implementation for \( f_1 \), \( f_2 \), and \( f_3 \) respectively. The speed-ups for multiplication are significantly higher. Moreover, MPS results are comparable with the C implementation of similar algorithms in BPAS. Figure 10 then highlights the efficiency of \( m \)-ary addition (see Section 5.1), compared to iterative applications of binary addition. Recall that \( m \)-ary addition is exploited in the Weierstrass preparation algorithm.

![Fig. 7: Computing \( \frac{1}{f} \) and \( \frac{1}{f} \cdot f \) for \( f_1 = 1 + X_1 + X_2 \).](image1.png)

![Fig. 8: Computing \( \frac{1}{f} \) and \( \frac{1}{f} \cdot f \) for \( f_2 = 1 + X_1 + X_2 + X_3 \).](image2.png)
Fig. 9: Computing $\frac{1}{f}$ and $\frac{1}{f} \cdot f$ for $f_3 = 2 + \frac{1}{3}(X_1 + X_2)$.

Fig. 10: Computing $f = \sum_{i=1}^{k} \frac{1}{1-x-y}$ using $m$-ary and binary addition.

Fig. 11: Computing Weierstrass preparation of $f_1 = \frac{1}{1+X_1+X_2}X_3^k + \cdots + X_2X_3 + X_1 \in \mathbb{K}[X_1,X_2][X_3]$.

Fig. 12: Computing Weierstrass preparation of $f_2 = \frac{1}{1+X_1+X_2}X_3^k + \cdots + X_2X_3 + X_1 \in \mathbb{K}[X_1,X_2][X_3]$.

Next, we compare the performance of Weierstrass preparation (Section 5.2). Figures 11 and 12 demonstrate the running time of this algorithm for two different UPoPS. Looking at these results, we can see a $2200 \times$ speed-up in comparison with the similar algorithm in RCPS and timings comparable to BPAS.

We also compare the factorization via Hensel's lemma and Taylor shift algorithms for a set of UPoPS $f = \prod_{i=1}^{k} (X_2 - i) + X_1(X_2^{k-1} + X_2)$ in $\mathbb{K}[X_1][X_2]$ with $k=3,4$ in Figures 13 and 14. Our factorization implementation is orders of magnitude faster than that of RCPS. However, factorization performs worse than expected compared to BPAS, having already seen comparable performance of Weierstrass preparation in Figures 11 and 12. This difference can be attributed...
to Taylor shift, the other core operation of HenselFactorize, as seen in Figure 14. The implementation in MPS is slower than the same procedure in BPAS by several order of magnitude. This, in turn, can be attributed to using Maple matrix arithmetic, rather than the direct manipulation of C-arrays as in BPAS, within the Taylor shift algorithm.

7 Conclusions and Future Work

Throughout this work we have discussed the object-oriented design and implementation of power series and univariate polynomials over power series following lazy evaluation techniques. Basic arithmetic operations for both are examined as well as Weierstrass Preparation Theorem, Taylor shift, and factorization via Hensel’s lemma for univariate polynomials over power series. Our implementation in MAPLE is orders of magnitude faster than the existing multivariate power series implementation in the PowerSeries package of the RegularChains library. Moreover, our implementation is comparable with the C implementation of power series and univariate polynomials over power series in BPAS.

Further work is needed to extend lazy evaluation techniques to more sophisticated algorithms. For example, a general Extended Hensel Construction (EHC) [13], and the Abhyankar-Jung Theorem [14]. As a consequence, it is possible to re-implement the EHC algorithm found in RegularChains using this library. Further, as MAPLE supports multithreading, it is possible to apply parallel processing to our algorithms. In particular, the computation of UPoPS coefficients in Weierstrass preparation is embarrassingly parallel. Meanwhile, the successive application of Weierstrass preparation and Taylor shift in HenselFactorize
present an opportunity for *pipelining*. Both should be exploited in to achieve even further performance improvements.

**Acknowledgements**

The authors would like to thank MITACS of Canada (award IT19704) and NSERC of Canada (award CGSD3-535362-2019).

**References**

1. [https://github.com/orcca-uwo/MultivariatePowerSeries](https://github.com/orcca-uwo/MultivariatePowerSeries) (2021)
2. Alvandi, P., Kazemi, M., Moreno Maza, M.: Computing limits with the regular-chains and powerseries libraries: From rational functions to zariski closure. ACM Communications in Computer Algebra **50**(3), 93–96 (2016)
3. Asadi, M., Brandt, A., Chen, C., Covano, S., Mansouri, F., Mohajerani, D., Moir, R.H.C., Moreno Maza, M., Wang, L.X., Xie, N., Xie, Y.: Basic Polynomial Algebra Subprograms (BPAS) (2020), [www.bpaslib.org](http://www.bpaslib.org)
4. Asadi, M., Brandt, A., Moir, R.H.C., Moreno Maza, M.: Algorithms and data structures for sparse polynomial arithmetic. Mathematics **7**(5), 441 (May 2019)
5. Bernardin, L., Chin, P., DeMarco, P., Geddes, K.O., Hare, D.E.G., Heal, K.M., Labahn, G., May, J.P., McCarron, J., Monagan, M.B., Ohashi, D., Vorkoetter, S.M.: Maple Programming Guide. Maplesoft, a division of Waterloo Maple Inc. (1996-2020)
6. Brandt, A., Kazemi, M., Moreno Maza, M.: Power series arithmetic with the BPAS library. In: Computer Algebra in Scientific Computing (CASC). pp. 108–128. Springer (2020)
7. Burge, W.H., Watt, S.M.: Infinite structures in scratchpad II. In: European Conference on Computer Algebra. pp. 138–148. Springer (1987)
8. Fischer, G.: Plane algebraic curves. American Mathematical Society (2001)
9. Harper, R.: Practical foundations for programming languages: Lazy Evaluation, pp. 323–332. Cambridge University Press, 2nd edn. (2016)
10. Karczmarczuk, J.: Generating power of lazy semantics. Theoretical Computer Science **187**(1-2), 203–219 (1997)
11. Maplesoft, a division of Waterloo Maple Inc.: Maple 2020, [www.maplesoft.com/](http://www.maplesoft.com/)
12. Monagan, M.B., Vrbik, P.: Lazy and forgetful polynomial arithmetic and applications. In: Computer Algebra in Scientific Computing, 11th International Workshop, CASC 2009, Proceedings. pp. 226–239 (2009)
13. Moreno Maza, M.: Polynomials over power series and their applications to limit computations (Tutorial at Computer Algebra in Scientific Computing (CASC), 2018), [www.csd.uwo.ca/~mmorenom/Publications/Polynomials_over_power_series_and_their_applications_lecture.PDF](http://www.csd.uwo.ca/~mmorenom/Publications/Polynomials_over_power_series_and_their_applications_lecture.PDF)
14. Parusiński, A., Rond, G.: The Abhyankar–Jung theorem. Journal of Algebra **365**, 29–41 (2012)
15. The Maple Developers: SolveTools package in Maple 2020, maplesoft, a division of Waterloo Maple Inc. [www.maplesoft.com/support/help/Maple/view.aspx?path=SolveTools](http://www.maplesoft.com/support/help/Maple/view.aspx?path=SolveTools)
16. The Sage Developers: SageMath, the Sage Mathematics Software System (2020), [www.sagemath.org](http://www.sagemath.org)
17. van der Hoeven, J.: Relax, but don’t be too lazy. Journal of Symbolic Computation 34(6), 479–542 (2002)
18. von zur Gathen, J., Gerhard, J.: Modern computer algebra. Cambridge University Press, NY, USA, 3 edn. (2013)
19. Wolfram Research Inc.: Mathematica (2020). www.wolfram.com/mathematica