Western Graduate&PostdoctoralStudies

Western University

Scholarship@Western

Electronic Thesis and Dissertation Repository

7-22-2020 4:00 PM

An Implementation of Power Series in the BPAS Library

Mahsa Kazeminooreddinvand, *The University of Western Ontario*

Supervisor: Moreno Maza, Marc, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

© Mahsa Kazeminooreddinvand 2020

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Recommended Citation

Kazeminooreddinvand, Mahsa, "An Implementation of Power Series in the BPAS Library" (2020). *Electronic Thesis and Dissertation Repository*. 7094.
https://ir.lib.uwo.ca/etd/7094

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

Abstract

We discuss the design and implementation of lazy multivariate power series, univariate polynomials over power series, and their associated arithmetic within the Basic Polynomial Algebra Subprograms (BPAS) Library. This implementation is employed by 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: Power series, Weierstrass preparation theorem, Hensel lemma, Factorization, C programming, High performance, Data structure, Bifurcation theory

Lay Summary

We explore the design and implementation of lazy multivariate power series, univariate polynomials with power series coefficients, and their associated arithmetic within the Basic Polynomial Algebra Subprograms (BPAS) Library. This implementation is employed by 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". We further address the application of these tools in bifurcation theory to analyze singularities of smooth maps.

Co-Authorship Statement

The results of this thesis are gathered from an accepted paper co-authored with Alexander Brandt and Marc Moreno Maza entitled "Power Series Arithmetic with the BPAS Library" and a publication co-authored with Marc Moreno Maza entitled "Detecting Singularities Using the PowerSeries Library ". The abstract, Chapters 1–6 and 8 are extracted from the former and the latter covers Chapter 7.

Acknowledgements

This thesis would not have been possible without support, help and love of many. First and foremost, I am forever grateful to my supervisor, Professor Marc Moreno Maza for his never-ending support, invaluable lessons, and precious guidance throughout the program as well as during the time I worked at ORCCA lab as a visiting scholar in 2015. Marc has provided me with many opportunities to learn, grow, pursue my interests and acquire hands-on experience. He has taught me the values of hard work, dedication and perseverance and I have had the privilege of contributing to several projects in his research group.

I am also deeply thankful to Alexander Brandt who has taught me many lessons, provided me with numerous technical supports and supervised me in the design and implementation of the results of this thesis in the `BPAS` library. His patience and unique method of teaching has allowed me to learn things at my own pace, make mistake and learn from them.

I would like to thank Dr. Robert Moir who has supervised me realizing a draft implementation of some of the results of this thesis in Python and also guided me in the early stages of my first internship at Maplesoft company.

I would like to thank my thesis committee members Professors Lucian Ilie, David Jeffrey, and Boyu Wang.

I would like to acknowledge Dr. John May and Dr. Erik Postma at Maplesoft Inc. for their lessons, guidance and support helping me to integrate the new features of the `RegularChains` library into Maple 2020. In particular, I very much appreciate Dr. John May for his supervision throughout the time I was working on the Rubi project.

I would also like to thank Ms. Janice Wiersma, Graduate Affairs Coordinator in Computer Science Department, for her kind help with various administrative tasks during my studies.

I would like to thank all my friends inside and outside of ORCCA lab for their kind help and support. I particularly would like to thank Davood who has been very supportive, helped me with many technical questions, generously shared his experiences with me on different topics in computer science. A special thank goes to my friend, Elham, who has always been there for me and given me encouragement.

Last but not least, my deepest love and gratitude goes to my parents, my siblings and my fiancé for their tremendous support, genuine encouragement and dedication.

To my wonderful parents for their endless love, encouragement and support

To my dearest, Mehdi, for his love, support and calm presence

Contents

Certificate of Examination **ii**

Abstract **ii**

Lay Summary **iii**

Co-Authorship Statement **iv**

Acknowledgements **v**

List of Figures **ix**

List of Tables **x**

1 Introduction **1**

2 Background **3**
- 2.1 Power Series 3
- 2.2 Weierstrass Preparation Theorem 7
- 2.3 Hensel Lemma 9

3 The Design and Implementation of Lazy Power Series **11**
- 3.1 The Power Series Data Structure, Generators, and Ancestors 13
- 3.2 Implementing Power Series Arithmetic 15
- 3.3 Benchmarks: Power Series Multiplication and Division 17

4 The Design and Implementation of Univariate Polynomials over Power Series **27**
- 4.1 UPOPS Taylor Shift 27

5 A Lazy Weierstrass Preparation **31**
- 5.1 Benchmarks: Weierstrass Preparation 34

6 A Lazy Factorization via Hensel's Lemma **44**
- 6.1 Benchmarks: Factorization via Hensel's Lemma 45

7 Applications in Bifurcation Theory **49**
- 7.1 Introduction 49
- 7.2 Background 51

7.2.1 Concepts from Singularity Theory 51
7.2.2 The Extended Hensel Construction 53
7.2.3 The PowerSeries Library 54
7.3 Applications 55
7.3.1 The Pitchfork Bifurcation 55
7.3.2 The Winged Cusp Bifurcation 56

8 Conclusions and Future Work **58**

Bibliography **58**

Curriculum Vitae **62**

List of Figures

3.1 Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 1 + X_1 + X_2$ 19
3.2 Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 1 + X_1 + X_2 + X_3$. 20
3.3 Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 2 + \frac{1}{3}(X_1 + X_2)$ 20

5.1 Applying Weierstrass preparation on family (i) for increasing precisions. 43
5.2 Applying Weierstrass preparation on family (ii) for increasing precisions. . . . 43

6.1 Applying factorization via Hensel's lemma to the UPOPS $f_1 = (Z-1)(Z-2)(Z-3) + X_1(Z^2+Z)$ and $f_2 = (Z-1)(Z-2)(Z-3)(Z-4) + X_1(Z^3+Z)$. . . 48

7.1 Figures (*a*) and (*b*) depict the bifurcation diagrams of g and NF(g), respectively. 52
7.2 On the right: Weierstrass Preparation Factorization for a univariate polynomial with multivariate power series coefficients. On the Left: Extended Hensel construction applied to a trivariate polynomial for computing its absolute factorization. 54
7.3 Extended Hensel construction applied to a bivariate polynomial for computing its Puiseux parametrizations around the origin. 54
7.4 EHC applied to $\Phi_1(x, y, \lambda)$. 56
7.5 Pitchfork bifurcation diagram associated with g in Equation (7.9). 56
7.6 The winged cusp bifurcation diagram. 56
7.7 EHC applied to $\Psi_2(x, y, \lambda)$. 57
7.8 Bifurcation diagram associated with g in (7.11). 57

List of Tables

3.1 A comparison of timings (with a time limit of 1800 seconds) for computing the inverse of the power series $2 + \frac{1}{3}(X_1 + X_2)$ to a precision between 100 and 2000. 21

3.2 A comparison of timings (with a time limit of 1800 seconds) for computing the inverse of the power series $1 + X_1 + X_2$ to a precision between 100 and 2000. 22

3.3 A comparison of timings (with a time limit of 1800 seconds) for computing the inverse of the power series $1 + X_1 + X_2 + X_3$ to a precision between 100 and 1200. 23

3.4 A comparison of timings (with a time limit of 1800 seconds) for multiplying $2+\frac{1}{3}(X_1+X_2)$ by its inverse and updating the terms based on the precision between 100 and 2000. 24

3.5 A comparison of timings (with a time limit of 1800 seconds) for multiplying $1+X_1+X_2$ by its inverse and updating the terms based on the precision between 100 and 2000. . . 25

3.6 A comparison of timings (with a time limit of 1800 seconds) for multiplying $1 + X_1 + X_2 + X_3$ by its inverse and updating the terms based on the precision between 100 and 1200. 26

5.1 A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^3 + Y^2 + X_2Y + X_1$ to a precision between 10 and 400. 35

5.2 A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^4 + Y^3 + Y^2 + X_2Y + X_1$ to a precision between 10 and 400. . . 36

5.3 A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^5 + Y^4 + Y^3 + Y^2 + X_2Y + X_1$ to a precision between 10 and 400. 37

5.4 A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^6 + Y^5 + Y^4 + Y^3 + Y^2 + X_2Y + X_1$ to a precision between 10 and 400. 38

5.5 A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^3 + X_2Y^2 + Y + X_1$ to a precision between 10 and 460. . . . 39

5.6 A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^4 + X_2Y^3 + Y^2 + Y + X_1$ to a precision between 10 and 420. . . 40

5.7 A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^5 + X_2Y^4 + Y^3 + Y^2 + Y + X_1$ to a precision between 10 and 400. 41

5.8 A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^6 + X_2Y^5 + Y^4 + Y^3 + Y^2 + Y + X_1$ to a precision between 10 and 400. 42

6.1 Factorization via Hensel for $(Z-1)(Z-2)(Z-3)+X_1(Z^2+Z)$ to a precision between 25 and 2000. Here, EHC and FVHL stand for the `ExtendedHenselConstruction` and `FactorizationViaHenselLemma` functions, respectively. Note that the benchmarks were collected with a time limit of 1800 seconds. 46
6.2 Factorization via Hensel for $(Z-1)(Z-2)(Z-3)(Z-4)+X_1(Z^3+Z)$ to a precision between 25 and 2000. Here, EHC and FVHL stand for the `ExtendedHenselConstruction` and `FactorizationViaHenselLemma` functions, respectively. Note that the benchmarks were collected with a time limit of 1800 seconds. 47

Chapter 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. Power series capture concepts and properties, such as that of a function limit, that cannot be described with notions and techniques from discrete mathematics. Of course, the fact that power series may have an infinite number of terms brings 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 [22] for polynomial system solving [10] based on Hensel lifting and its variants [34]. 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*, and 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 computed or "produced" concurrent with being "consumed" in subsequent computations. These procedure calls can then be seen as the stages of a *pipelined* computation [23, Ch. 9].

In this thesis, we present our implementation of *multivariate power series* (Chapter 3) and *univariate polynomials over multivariate power series* "UPOPS" (Chapter 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) [2, 27] and the Jung-Abhyankar Theorem [25], is our driving application. We discuss a lazy implementation of factoring via Hensel's lemma (Chapter 6) by means of lazy Weierstrass preparation (Chapter 5). We further highlight one of the applications of the EHC method, which is implemented in MAPLE's `PowerSeries` library, in bifurcation theory to determine singularities of smooth maps (Chapter 7). Finally, we summarize the results of this thesis along with a future outlook for our design and implementation (Chapter 8).

Our implementation is publicly available as part of the Basic Polynomial Algebra Subprograms (BPAS) library [6], 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 of multivariate power series and UPOPS is both sequential and over the field of rational numbers. However, the BPAS library has the necessary infrastructure, in particular asynchronous generators, see [7], 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` library [5, 20] and SAGEMATH [30]. 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 [19] (discussing different mathematical objects with an infinite length of data) and the work of Monagan and Vrbik [24] (discussing sparse polynomial arithmetic). Lazy univariate power series, in particular, have been implemented by Burge and Watt [9] and by van der Hoeven [32]. However, up to our knowledge, our implementation is the first for *multivariate* power series in a compiled code.

This thesis is a joint work with Alexander Brandt and Marc Moreno Maza; see [1, 20].

Chapter 2

Background

This chapter introduces algebraic concepts, structures and terminologies required throghout this thesis. We start with formal power series arithmetic and deal with arithmetic in the context of univariate polynomials with multivariate power series coefficients. Next, we discuss the Weierstrass Preparation Theorem providing the foundation for factorizing univariate polynomials over power series. We end this chapter with a theoretical discussion of Hensel Lemma which utilizes the above mentioned Theorem in order to factor univariate polynomials over power series into linear factors.

2.1 Power Series

Definition 1 *A non-empty set R with two binary operations called addition $(+)$ and multiplication $(\cdot)$ is called a commutative ring if it satisfies the following conditions*

(i) *for all $x, y \in R$, $x + y = y + x$,*

(ii) *for all $x, y, z \in R$, $(x + y) + z = x + (y + z)$,*

(iii) *there exists an element $0 \in R$ such that $0 + x = x + 0 = x$ for all $x \in R$,*

(iv) *for every $x \in R$ there exists $-x \in R$ such that $x + (-x) = (-x) + x = 0$,*

(v) *for all $x, y \in R$, $x \cdot y = y \cdot x$,*

(vi) *for all $x, y, z \in R$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$,*

(vii) *there exists an element $0 \neq 1 \in R$ such that $1 \cdot x = x \cdot 1 = x$ for all $x \in R$,*

$(viii)$ *for all $x, y, z \in R$, $x \cdot (y + z) = x \cdot y + x \cdot z$.*

Definition 2 *Let R be a commutative ring. A subset I of R is called an ideal when*

(i) *$0 \in I$,*

(ii) *if $x, y \in I$ then $x + y \in I$,*

(iii) *if* $x \in I$ *then* $-x \in I$,

(iv) *if* $r \in R$ *and* $x \in I$ *then both* $r \cdot x \in I$ *and* $x \cdot r \in I$.

Definition 3 *Let* R *be a commutative ring and* $\{I_i\}_{i=1}^{\infty}$ *be an increasing sequence of ideals in* R*, that is*

$$I_1 \subseteq \cdots \subseteq I_{k-1} \subseteq I_k \subseteq I_{k+1} \subseteq \cdots$$

We say that R *is a Noetherian ring if the sequence is ultimately constant that is there exists* $n \in \mathbb{N}$ *such that* $I_n = I_{n+1} = \cdots$.

Definition 4 *Let* R *be a commutative ring. A non-zero element* $a \in R$ *is called a zero divisor if* $a \cdot b = 0$ *is valid for some non-zero element* $b \in R$*. The ring* R *is called an integral domain if it has no zero divisor.*

Definition 5 *Assume that* R *is a commutative ring. A non-zero and non-unit element* $p \in R$ *is called a prime when for some* a, b *inR if* p *divides* $a \cdot b$ *then* p *divides* a *or* p *divides* b*. An integral domain in which every non-zero non-unit element is represented by a product of prime elements is called a unique factorization domain (UFD).*

Definition 6 *If for every* $0 \neq x \in R$ *there exists an element* $\frac{1}{x} \in R$ *such that* $x \cdot (\frac{1}{x}) = (\frac{1}{x}) \cdot x = 1$ *then* R *is called a field. Here, a field is denoted by* F.

Definition 7 (i) *A sequence* $x_1, x_2, x_3, \ldots$ *in* F converges *to* x *if for all* $0 < \epsilon \in F$ *there exists* $N \in \mathbb{N}$ *s.t. for all* $n \in \mathbb{N}$ *we have* $n \geq N \Rightarrow \mid x - x_n \mid < \epsilon$.

(ii) *A sequence* $x_1, x_2, x_3, \ldots$ *in* F *is called a Cauchy sequence if for all* $0 < \epsilon \in F$ *there exists* $N \in \mathbb{N}$ *s.t. for all* $m, n \in \mathbb{N}$ *we have* $m, n > N \Rightarrow \mid x_m - x_n \mid < \epsilon$.

Definition 8 *A field* F *is called complete if every Cauchy sequence in* F *converges to an element of* F.

Definition 9 *Let* R *be a commutative ring.*

(i) *The set* $R^{\mathbb{N}}$ *of all infinite sequences of elements of* R *indexed by* $\mathbb{N} \cup \{0\}$ *constructs the univariate power series ring* $R[[X]]$.

(ii) *The ring of multivariate power series in variables* $X_1, \ldots, X_n$ *and with coefficients in* R *is denoted by* $R[[X_1, \ldots, X_n]]$ *and defined recursively as* $R[[X_1]]$ *if* $n = 1$ *and as* $(R[[X_1, ..., X_{n-1}]])[[X_n]]$ *otherwise.*

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

$$f_{(d)} = \sum_{|e|=d} a_e X^e \quad \text{and} \quad f^{(d)} = \sum_{k \leq d} f_{(k)}. \tag{2.1}$$

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 $a_e \in R$ holds.

Definition 10 *Let $f, g \in R[[X_1, \ldots, X_n]]$. Then the sum, difference, and product of f and g are given by*

$$\begin{aligned} f \pm g &= \sum\nolimits_{d\in\mathbb{N}} (f_{(d)} \pm g_{(d)}) \\ fg &= \sum\nolimits_{d\in\mathbb{N}} (\Sigma_{k+\ell=d}\, (f_{(k)} g_{(\ell)})) \,. \end{aligned}$$

We consider the following example to clarify this Definition.

Example 1 *Let*

$$f = \sum_{(e_1,e_2)} (2e_1+1)(-1)^{e_1}(2e_2+1)^{e_1} X_1^{e_1} X_2^{e_2}, \quad g = \sum_{(e_1,e_2)} (2e_1+2)(e_2) X_1^{2e_1+1} X_2^{e_2}.$$

It follows from (2.1) *that*

$$f_{(0)} = 1, \quad f_{(1)} = 3X_2 - 3X_1, \quad f_{(2)} = 5X_1^2 + 5X_2^2 - 9X_1X_2, \ldots$$

and

$$g_{(0)} = 0, \quad g_{(1)} = 0, \quad g_{(2)} = 2X_1X_2, \ldots .$$

Hence,

$$\begin{aligned} f \pm g &= (f_{(0)} \pm g_{(0)}) + (f_{(1)} \pm g_{(1)}) + (f_{(2)} \pm g_{(2)}) + \cdots \\ &= 1 + (3X_2 - 3X_1) + (5X_1^2 + 5X_2^2 - 9X_1X_2 \pm 2X_1X_2) + \cdots . \end{aligned}$$

and

$$\begin{aligned} fg &= (f_{(0)}g_{(0)}) + (f_{(0)}g_{(1)} + f_{(1)}g_{(0)}) + (f_{(0)}g_{(2)} + f_{(1)}g_{(1)} + f_{(2)}g_{(0)}) + \cdots \\ &= 2X_1X_2 + \cdots . \end{aligned}$$

Remark 1 *Note that the arithmetic associated with univariate polynomials with power series coefficients is inherited from that of power series.*

Example 2 *Let $f = X_2^2 + Y^2$, $g = 1 + X_1^3 + 3X_2Y^2 + (1 + X_1X_3)Y^6 \in R[[X_1, X_2, X_3]][Y]$ be univariate polynomials in Y with power series coefficients in $R[[X_1, X_2, X_3]]$. Then*

$$\begin{aligned} f \pm g &= X_2^2 \pm 1 + Y^2 \pm (1 + X_1X_3)Y^6 \\ fg &= X_2^2 + Y^2 + X_2^2Y^6 \end{aligned}$$

Definition 11 *The order of a formal power series $f \in R[[X_1, \ldots, X_n]]$ is defined as:*

$$\operatorname{ord}(f) = \begin{cases} \min\{d \mid f_{(d)} \neq 0\} & \text{if } f \neq 0 \\ \infty & \text{if } f = 0. \end{cases}$$

Remark 2 *For $f, g \in R[[X_1, \ldots, X_n]]$, we have*

$$\operatorname{ord}(f+g) \geq \min\{\operatorname{ord}(f), \operatorname{ord}(g)\} \text{ and } \operatorname{ord}(fg) = \operatorname{ord}(f) + \operatorname{ord}(g).$$

Now we list several properties of the ring of formal power series.

(a) $R[[X_1, \ldots, X_n]]$ is an integral domain,

(b) the set $\mathcal{M} = \{f \in R[[X_1, \ldots, X_n]] \mid \operatorname{ord}(f) \geq 1\}$ is the only maximal ideal of $R[[X_1, \ldots, X_n]]$,

(c) for all $k \in \mathbb{N}$, we have $\mathcal{M}^k = \{f \in R[[X_1, \ldots, X_n]] \mid \operatorname{ord}(f) \geq k\}$.

Definition 12 (Krull Topology) *Let $(f_n)_{n\in\mathbb{N}}$ be a sequence of elements of $R[[X_1, \ldots, X_n]]$ and let $f \in R[[X_1, \ldots, X_n]]$. 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 $n \geq N \Rightarrow f - f_n \in \mathcal{M}^k$,*
- $(f_n)_{n\in\mathbb{N}}$ *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 $n, m \geq N \Rightarrow f_m - f_n \in \mathcal{M}^k$.*

Proposition 1

- *We have $\bigcap_{k\in\mathbb{N}} \mathcal{M}^k = \langle 0 \rangle$*
- *If every Cauchy sequence in R converges, then every Cauchy sequence of $R[[X_1, \ldots, X_n]]$ converges too.*

Proposition 2 *For all $f \in R[[X_1, \ldots, X_n]]$, the following properties are equivalent*

(i) *f is a unit*

(ii) $\operatorname{ord}(f) = 0$

(iii) *$f \notin \mathcal{M}$.*

PROOF. This follows from the classical observation that for $g \in R[[X_1, \ldots, X_n]]$, with $\operatorname{ord}(g) > 0$, the following holds in $R[[X_1, \ldots, X_n]]$

$$(1-g)(1+g+g^2+\cdots) = 1$$

Since $(1+g+g^2+\cdots)$ is in fact a sequence of elements in $R[[X_1, \ldots, X_n]]$, proving the above relation formally requires the use of Krull Topology. □

Definition 13 *Let $\mathbb{K}$ be a complete field. Consider the formal power series $f = \sum_e a_e X^e$, $g = \sum_e b_e X^e$, $h = \sum_e c_e X^e \in \mathbb{K}[[X_1, \ldots, X_n]]$. Provided that the power series g is invertible. The quotient $h = \frac{f}{g}$ is defined as follows*

$$\frac{f}{g} = \frac{\sum_e a_e X^e}{\sum_e b_e X^e} = \sum_e c_e X^e.$$

In order to compute h, one can multiply the inverse of g by f or equate the coefficients in $f = gh$; that is

$$c_e = \frac{1}{b_0}(a_e - \sum_{k=1}^{\infty} b_k c_{e-k}).$$

The above Definition is illustrated by the following examples.

Example 3 (*a*) *Let* $f = 1$ *and* $g = 1 - 2X_1 + X_1^2$. *Then*

$$h = \frac{f}{g} = 1 + 2X_1 + 3X_1^2 + 4X_1^3 + 5X_1^4 + \cdots .$$

(*b*) *Let* $f = 1 - X_4 - X_1$ *and* $g = -2X_1X_4 + X_1^2 + X_2X_3 + X_4^4 + 1$. *Then*

$$h = \frac{f}{g} = 1 - X_1 - X_4 - X_1^2 + 2X_1X_4 - X_2X_3 + X_1^3 - X_1^2X_4 + X_1X_2X_3 - 2X_1X_4^2 + X_2X_3X_4 + \cdots .$$

2.2 Weierstrass Preparation Theorem

Consider a univariate polynomial with multivariate power series coefficients which is not identical to zero after evaluating all coefficients at the origin. The Weierstrass Preparation Theorem is concerned with finding a unique representation for this polynomial in the vicinity of the origin. This representation is given by two univariate polynomials with multivariate power series coefficients, one is known as the Weierstrass polynomial and the other one is a unit whose product gives the original polynomial.

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* $f_1, \ldots, f_r, h_0, \ldots, h_{r-1}$.

PROOF. Lemma 1 is essential to our implementation of Weierstrass Preparation Theorem (WPT). Hence, we give a proof by induction on r. Since $g_0h_0 = f_0 = 0$ 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 $\mathcal{M}^{r+1}$:

$$f_1 + f_2 + \cdots + f_r = g_1h_0 + (g_2h_0 + g_1h_1) + \cdots + (g_rh_0 + g_{r-1}h_1 + \cdots + g_1h_{r-1}).$$

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

Example 4 *Suppose that* $f = X_2$, $g = X_2 + X_1X_2^2$, $h = 1 - X_1X_2 \in \mathbb{K}[[X_1, X_2]]$. *As can be seen, modulo* $\mathcal{M}^3$

$$X_2 = (X_2 + X_1X_2^2)(1 - X_1X_2)$$

Using Lemma 1 we now attempt to derive g_1, g_2 *and* g_3

$$\begin{aligned} f_1 &= g_1h_0 \Longrightarrow g_1 = X_2 \\ f_2 &= g_2h_0 + g_1h_1 \Longrightarrow g_2 = 0 \\ f_3 &= g_3h_0 + g_2h_1 + g_1h_2 \Longrightarrow g_3 = 0 - (X_2)(-X_1X_2) = X_1X_2^2. \end{aligned}$$

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 $i \in \mathbb{N}$. 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 2.2.1 *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 $p = X_n^d + b_{d-1}X_n^{d-1} + \cdots + b_1 X_n + b_0$, 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.

We make three remarks before discussing the proof. 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.

PROOF. We give a proof of WPT, as this supports our implementation. 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 $\alpha = \sum_{i=0}^{\infty} c_i X_n^i$ 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:

$$
\begin{array}{rcl}
a_0 & = & b_0 c_0 \\
a_1 & = & b_0 c_1 + b_1 c_0 \\
a_2 & = & b_0 c_2 + b_1 c_1 + b_2 c_0 \\
 & \vdots & \\
a_{d-1} & = & b_0 c_{d-1} + b_1 c_{d-2} + \cdots + \cdots + b_{d-2} c_1 + b_{d-1} c_0 \\
a_d & = & b_0 c_d + b_1 c_{d-1} + \cdots + \cdots + b_{d-1} c_1 + c_0 \\
a_{d+1} & = & b_0 c_{d+1} + b_1 c_d + \cdots + \cdots + b_{d-1} c_2 + c_1 \\
 & \vdots & \\
a_{d+m} & = & b_0 c_{d+m} + b_1 c_{d+m-1} + \cdots + \cdots + b_{d-1} c_{m+1} + c_m \\
 & \vdots &
\end{array}
$$

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 $i \geq 0$. □

Example 5 (*a*) *Let* $f = X_2 + X_3 + X_1X_3^2 \in \mathbb{K}[[X_1, X_2]][X_3]$ *and* $\mathcal{M} = \langle X_1, X_2\rangle$. *Applying Theorem 2.2.1, modulo* $\mathcal{M}^{11}$*, gives rise to*

$$\begin{aligned} p &= X_2 + X_2^2X_1 + 2X_2^3X_1^2 + 5X_2^4X_1^3 + 14X_2^5X_1^4 + X_3 \\ \alpha &= 1 - X_2X_1 - X_2^2X_1^2 - 2X_2^3X_1^3 - 5X_2^4X_1^4 - 14X_2^5X_1^5 + X_1X_3. \end{aligned}$$

(*b*) *Let* $f = X_1^2 + X_2^2 + (X_2 + 1)X_3^2 + X_3^3 \in \mathbb{K}[[X_1, X_2]][X_3]$ *and* $\mathcal{M} = \langle X_1, X_2\rangle$. *Applying Theorem 2.2.1, modulo* $\mathcal{M}^5$*, results in*

$$\begin{aligned} p &= X_2^2 + X_1^2 - X_2^3 - X_2X_1^2 - X_2^2X_1^2 - X_1^4 + (-X_2^2 - X_1^2 + 2X_2^3 + 2X_2X_1^2 - X_2^4 + X_2^2X_1^2 + 2X_1^4)X_3 + X_3^2 \\ \alpha &= 1 + X_2 + X_2^2 + X_1^2 - 2X_2^3 - 2X_2X_1^2 + X_2^4 - X_2^2X_1^2 - 2X_1^4 + X_3. \end{aligned}$$

2.3 Hensel Lemma

Broadly speaking, there are two main flavors of Hensel's Lemma. The most formal, which is the focus of our study, treats univariate polynomials with multivariate power series coefficients with the aim of decomposing them into linear factors. In the second one, the multivariate power series coefficients are truncated up to a certain degree reducing the problem to univariate polynomials with polynomial coefficients. The applications of the latter can be found in factorization of polynomials, polynomials GCD and etc. Now let $f = a_kY^k + \cdots + a_1Y + a_0$ 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* meaning that every non-constant element of $\mathbb{K}[[X_1, \ldots, X_n]]$ has a root in $\mathbb{K}$. 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 2.3.1 (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:

$$\begin{aligned} g(X_1, \ldots, X_n, Y) &= f(X_1, \ldots, X_n, Y + c_r) \\ &= (Y + c_r)^k + a_1(Y + c_r)^{k-1} + \cdots + a_k \end{aligned}$$

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 $f_1, \ldots, f_{r-1}$. □

Example 6 (*a*) *Let* $f = Y^2 - 9Y + X_1 \in \mathbb{K}[[X_1]][Y]$ *and* $\mathcal{M} = \langle X_1 \rangle$. *Note that*

$$\overline{f} = Y^2 - 9Y = (Y - 9)(Y)$$

Theorem 2.3.1 provides the following factors modulo $\mathcal{M}^6$

$$\begin{aligned} f_1 &= -9 + \frac{1}{9}X_1 + \frac{1}{729}X_1^2 + \frac{2}{59049}X_1^3 + \frac{5}{4782969}X_1^4 + \frac{14}{387420489}X_1^5 + Y \\ f_2 &= \frac{-1}{9}X_1 - \frac{1}{729}X_1^2 - \frac{2}{59049}X_1^3 - \frac{5}{4782969}X_1^4 - \frac{14}{387420489}X_1^5 + Y. \end{aligned}$$

(*b*) *Let* $f = Y^4 + (4 + X_1)Y^3 - 7Y^2 - 10Y \in \mathbb{K}[[X_1]][Y]$ *and* $\mathcal{M} = \langle X_1 \rangle$. *Observe that*

$$\overline{f} = Y^4 + 4Y^3 - 7Y^2 - 10Y = Y(Y + 5)(Y - 2)(Y + 1)$$

Applying Theorem 2.3.1, modulo $\mathcal{M}^6$, *gives*

$$\begin{aligned} f_1 &= Y \\ f_2 &= 5 + \frac{25}{28}X_1 + \frac{125}{21952}X_1^2 + \frac{9375}{8605184}X_1^3 - \frac{6896875}{13492928512}X_1^4 + \frac{110046875}{755603996672}X_1^5 + Y \\ f_3 &= -2 + \frac{4}{21}X_1 - \frac{176}{9261}X_1^2 + \frac{6416}{4084101}X_1^3 - \frac{159424}{1801088541}X_1^4 + \frac{10880}{113468578083}X_1^5 + Y \\ f_4 &= 1 - \frac{1}{12}X_1 + \frac{23}{1728}X_1^2 - \frac{331}{124416}X_1^3 + \frac{21487}{35831808}X_1^4 - \frac{375985}{2579890176}X_1^5 + Y. \end{aligned}$$

Chapter 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 generating function—a function which returns new terms for a power series to increase its precision. Such a generating function, or *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 generating 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 (in contrast to an object interface in object-oriented design). This interface has two main functions: one to return the homogeneous part of a power series, and one to return the polynomial part of a power series, each for a requested degree. These functions call some underlying generator to produce terms until the requested degree is satisfied.

Two such functions are shown using MAPLE-like pseudo-code in Listing 3.1 as `homog_part_ps` and `polynomial_part_ps`, respectively. The key element to these functions are their automatic calls to the generating function `GEN` if the requested degree is greater than the current degree of the power series.

As a first example, consider, the construction of the geometric series as a lazy power series, in MAPLE-style pseudo-code in Listing 3.2. A power series is a data structure holding a polynomial, a generating 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

```
homog_part_ps := proc(ps, d::integer)
    if (d > ps[DEG]) then
        for i from ps[DEG] + 1 to d do
            ps[POLY] := ps[POLY] + ps[GEN](i)
        end do;
    end if;
    return homogeneous_part(ps[POLY], d);
end proc;

polynomial_part_ps := proc(ps, d::integer)
    if (d > ps[DEG]) then
        for i from ps[DEG] + 1 to d do
            ps[POLY] := ps[POLY] + ps[GEN](i)
        end do;
    end if;
    return truncate_poly(ps[POLY], d);
end proc;
```

Listing 3.1: A lazy power series design where a generating function is called on demand through some top-level functional interface.

```
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 3.2: The geometric series as a lazy power series.

dynamic typing and first-class function objects support such manipulation. This manipulation is further interesting where power series can also be created through manipulation of existing power series, and thus requires generating functions to be dynamically constructed from existing power series and their generating functions. For example, when a power series is created by a arithmetic operation applied to two existing power series.

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 [8], but on the other hand creates an impressive design challenge to effectively handle the need for dynamic function manipulation. In this chapter we detail our resulting solution, which makes use of a so-called *ancestry* in order for the generating function of a newly created power series to “remember” from where it came. We begin in Section 3.1 with an overview of the basic power series representation, its data structure, and our solution to generating functions in C. In Section 3.2 we discuss power series multiplication and division, thus discussing the combination of this data structure with our run-time support for creating a new generator dynamically. Lastly, Section 3.3 presents experimentation of our implementation against SAGEMATH and MAPLE showing orders of magnitude improvement in computation time.

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 generating 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 generating function (see, e.g., [29, 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 parameters. 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 generating 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 resulting from an arithmetic operation.

Therefore, to introduce some flexibility in the generating functions, we extend the previous definition of a generating function to include a finite number of `void` pointer parameters following the first integer parameter. These additional parameters are to be used by the generating function to assist in generating the homogeneous polynomial to return. 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. `int` or `float`). In 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 3.5.

Our implementation, which supports power series arithmetic, Weierstrass preparation, and factorization via Hensel's lemma, currently requires only 4 unique 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

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

call the number of these `void` pointer parameter the generator's *order*. We have thus *nullary generators*, *unary generators*, *binary generators*, and *tertiary generators*. 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 in order to choose the correct generator from the union type to call 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 3.3. 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. Whereas `homog_part_ps` and `polynomial_part_ps` in the pseudo-code of Listing 3.1 used generator function objects generically, our functions, simply use function pointers rather than function objects.

In general, these void pointer generator parameters are actually pointers to existing power series structs. For example, the operands of an arithmetic operation would become parameters 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, an 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 (Chapter 5) and factorization via Hensel's lemma (Chapter 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 (by being set as the value of the child's generator parameter) its reference count is incremented. Therefore, the user may "free" a power series when they are finished with it, and yet the memory persists as long as some other power series has reference to it. Therefore, a free in this case is merely a decrement of a reference counter. When the counter falls to 0, the data is actually freed, and moreover, upon being freed, a child power series will also decrement the reference count of its parents, since that reference has finally been removed.

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 3.4. We implement this as an enumeration instead of a Boolean so that the implementation is extensible to further parameter types. One may think that storing both a void pointer, along with an enumeration value which encodes the actual type of that pointer, to be wasteful. However, the additional memory usage is minimal compared to that used by the polynomial data itself. Moreover, alternative solutions using, for example, union types, would still need a way of determining the current valid field in the union for a particular context.

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 2.1, the power series product of $f, g \in \mathbb{K}[[X_1, \ldots, X_n]]$ is defined simply as $h = fg = \sum_{d \in \mathbb{N}} (\Sigma_{k+\ell=d} (f_{(k)} g_{(\ell)}))$. 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 3.5. 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

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

// An updated PowerSeries struct with reference counts and parameter types.
typedef struct PowerSeries {
    int deg;
    int alloc;
    Poly_ptr* homog_polys;
    HomogPartGenerator_u gen;
    int genOrder;
    int refcont;
    void *genParam1, *genParam2, *genParam3;
    GenParamType_e paramType1, paramType2, paramType3;
} PowerSeries_t;

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 3.4: Extending the power series struct to include reference counting (as the `refCount` field) and management of reference counts via `destroyPowerSeries_PS`.

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

$$c_e = \frac{1}{b_0}\left(a_e - \sum_{k+\ell=e} b_k c_\ell\right).$$

This formula can easily be rearranged in order to find the homogeneous part of h for a given degree d:

$$h_{(d)} = \frac{1}{g_{(0)}}\left(f_{(d)} - \sum_{k=1}^{d} g_{(k)} h_{(d-k)}\right).$$

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 division in $\mathbb{K}$, which is guaranteed since g is invertible and thus $g_{(0)} \neq 0$. The rest follows by induction.

In our graded representation, where power series are updated with successive homogeneous parts, this formula yields a generator for a power series quotient. The realization of this generator in code is simple, as shown in Listing 3.6. This code, like all power series operations, follows a symmetric pattern to power series multiplication. The division operation `dividePowerSeries_PS` merely allocates a power series and initializes it lazily with a single term. The void generator is `homogPartVoid_quo_PS` which calls the true generator

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

`homogPart_quo_PS`. 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.

3.3 Benchmarks: Power Series Multiplication and Division

We now look to compare our implementation against SageMath [30], and Maple 2020. In Maple 2020, the unexposed `PowerSeries` library [5, 20] provides lazy multivariate power series, meanwhile the built-in (and thus exposed) `mtaylor` command provides *truncated* taylor series expansions. 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 3.1–3.3.

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 [11] underlies the

```
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 dividePolynomials(s, homogPart(0, g))
}

Poly_ptr homogPartVoid_quo_PS(int d, void* p1, void* p2, void* p3) {
    return homogPart_prod_PS(d, (PowerSeries_t*) p1,
                                (PowerSeries_t*) p2,
                                (PowerSeries_t*) p3);
}

PowerSeries_t* dividePowerSeries_PS(PowerSeries_t* f, PowerSeries_t* g) {
    if(!isInvertible_PS(g)) {
        exit(DIVISION_BY_ZERO)
    }

    reserve_PS(f); reserve_PS(g);
    PowerSeries_t* quo = allocPowerSeries(1);
    quo->gen.ternaryGen= &(homogPartVoid_quo_PS)
    quo->genParam1 = (void*) f;
    quo->genParam2 = (void*) g;
    quo->genParam3 = (void*) quo;
    quo->paramType1 = POWER_SERIES;
    quo->paramType2 = POWER_SERIES;
    quo->paramType3 = PLAIN_DATA; //abuse this to avoid chicken-and-egg
    quo->deg = 0;
    quo->homogPolys[0] = homogPart_quo_PS(0, f, g, quo);
    return quo;
}
```

Listing 3.6: Computing the division of two power series, where `homogPart_quo_PS` is the generator of the quotient.

multivariate polynomial ring S while Flint [18] underlies the univariate polynomials used in univariate power series. Python 3.7 interfaces and joins these underlying implementations. To see exactly 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 thesis 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. Tables 3.1– 3.6 contain the experimental data for our comparison of power series multiplication and inversion in MAPLE 's `PowerSeries` library, SAGEMATH and BPAS. Figures 3.1–3.3 gather data from Tables 3.1– 3.6 in a graphical manner.

The first set of benchmarks are presented in Figure 3.1 where the power series $f = 1 + X_1 + X_2$ is both inverted and multiplied by its inverse. Figures 3.2 and 3.3 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. Note that power series arithmetic densify computations

and many other examples have been tried too. 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 [8], 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`.

The current experimentation deals with power series obtained by applying power series arithmetic operations to multivariate polynomials. We note that that both MAPLE 's `PowerSeries` library and its `BPAS` counterpart allow the creation of a power series from the function returning the sum of its terms of a given degree. Therefore, power series corresponding to analytic functions (around a given point) such as exponential and trigonometric functions can be manipulated by MAPLE 's `PowerSeries` library and its `BPAS` counterpart.

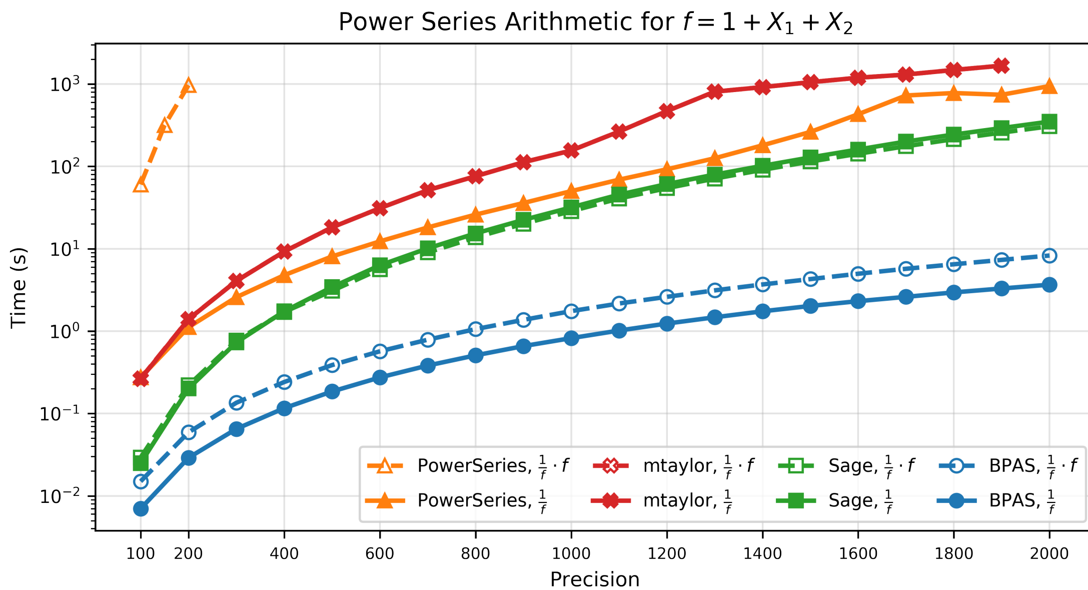

Figure 3.1: Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 1 + X_1 + X_2$

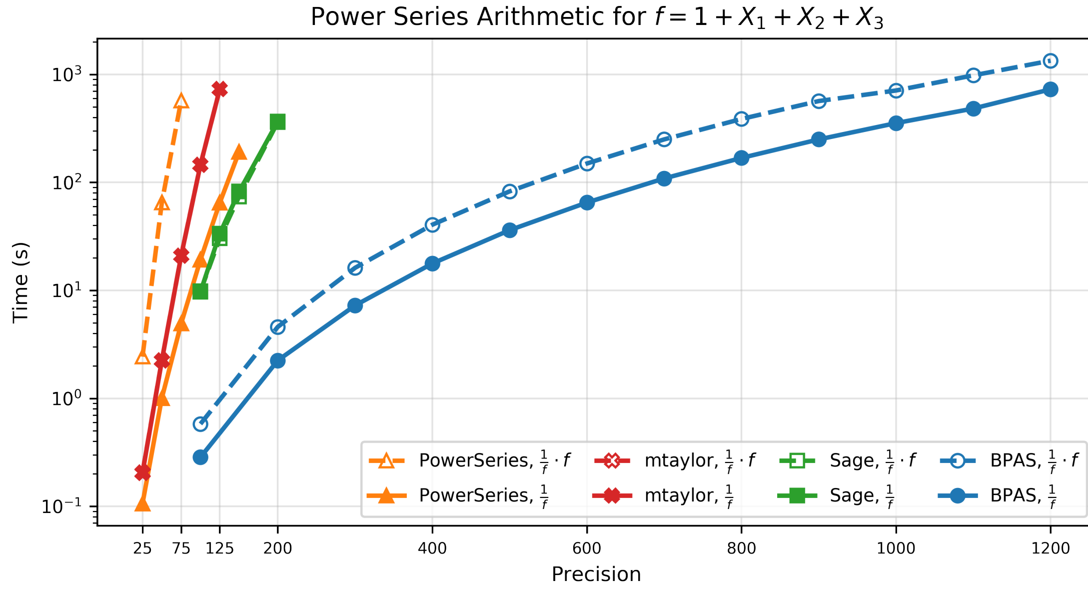

Figure 3.2: Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 1 + X_1 + X_2 + X_3$.

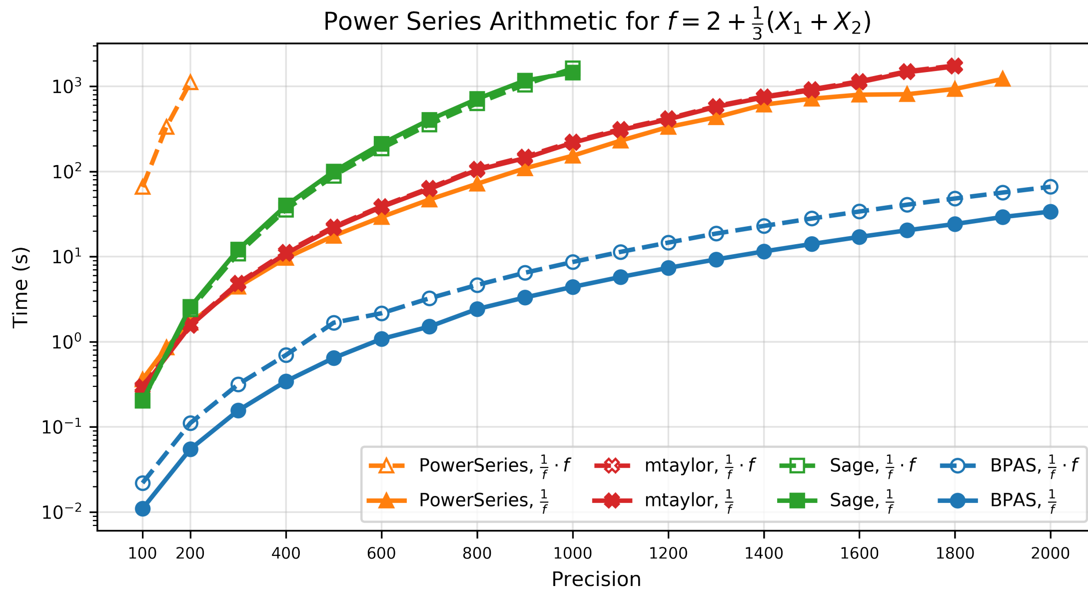

Figure 3.3: Computing $\frac{1}{f}$ and $f \cdot \frac{1}{f}$ for $f = 2 + \frac{1}{3}(X_1 + X_2)$

Precision	BPAS	SAGE	MAPLE
100	0.011	0.204	0.361
150	0.027	0.895	0.856
200	0.055	2.565	1.673
300	0.156	12.091	4.433
400	0.344	39.724	9.634
500	0.646	99.996	17.548
600	1.080	211.187	29.167
700	1.506	403.023	46.710
800	2.425	710.182	71.584
900	3.314	1167.297	108.798
1000	4.409	broken pipe	152.951
1100	5.761	broken pipe	229.669
1200	7.379	broken pipe	332.585
1300	9.271	broken pipe	432.806
1400	11.511	broken pipe	608.408
1500	14.055	broken pipe	714.256
1600	17.039	broken pipe	796.811
1700	20.373	broken pipe	806.888
1800	24.159	broken pipe	930.397
1900	29.203	broken pipe	1220.259
2000	33.781	broken pipe	1295.920 (broken pipe)

Table 3.1: A comparison of timings (with a time limit of 1800 seconds) for computing the inverse of the power series $2 + \frac{1}{3}(X_1 + X_2)$ to a precision between 100 and 2000.

Precision	BPAS	SAGE	MAPLE
100	0.007	0.025	0.274
200	0.029	0.201	1.116
300	0.065	0.729	2.582
400	0.116	1.739	4.772
500	0.185	3.421	8.071
600	0.274	6.297	12.294
700	0.382	10.109	18.152
800	0.508	15.357	25.957
900	0.658	22.407	35.927
1000	0.822	31.804	50.107
1100	1.017	45.404	69.030
1200	1.236	61.040	92.274
1300	1.473	79.589	125.599
1400	1.742	101.757	180.372
1500	2.022	128.574	262.254
1600	2.312	160.460	430.225
1700	2.608	200.519	726.641
1800	2.954	243.718	778.142
1900	3.295	292.257	740.904
2000	3.666	349.944	945.301

Table 3.2: A comparison of timings (with a time limit of 1800 seconds) for computing the inverse of the power series $1 + X_1 + X_2$ to a precision between 100 and 2000.

Precision	BPAS	SAGE	MAPLE
25			0.106
50			1.006
75			4.928
100	0.285	9.713	19.232
125		33.604	64.895
150		82.300	191.896
200	2.245	366.490	NaN
300	7.250	NaN	NaN
400	17.747	NaN	NaN
500	36.123	NaN	NaN
600	65.098	NaN	NaN
700	108.834	NaN	NaN
800	168.567	NaN	NaN
900	250.236	NaN	NaN
1000	354.211	NaN	NaN
1100	483.245	NaN	NaN
1200	728.184	NaN	NaN

Table 3.3: A comparison of timings (with a time limit of 1800 seconds) for computing the inverse of the power series $1 + X_1 + X_2 + X_3$ to a precision between 100 and 1200.

Precision	BPAS	SAGE	MAPLE
50	0.008	0.023	4.414
100	0.022	0.225	65.688
150	0.064	0.965	333.282
200	0.111	2.401	1110.165
300	0.316	10.847	5999.789 (broken pipe)
400	0.699	35.605	
500	1.675	88.789	
600	2.163	185.971	
700	3.236	356.137	
800	4.646	629.802	
900	6.465	1045.676	
1000	8.644	1622.464	
1100	11.369	broken pipe	
1200	14.616	broken pipe	
1300	18.653	broken pipe	
1400	22.924	broken pipe	
1500	28.029	broken pipe	
1600	33.868	broken pipe	
1700	40.708	broken pipe	
1800	48.148	broken pipe	
1900	56.575	broken pipe	
2000	66.104	broken pipe	

Table 3.4: A comparison of timings (with a time limit of 1800 seconds) for multiplying $2 + \frac{1}{3}(X_1 + X_2)$ by its inverse and updating the terms based on the precision between 100 and 2000.

Precision	BPAS	SAGE	MAPLE
100	0.015	0.0293	60.344
150			316.485
200	0.059	0.2200	969.457
300	0.135	0.7710	NAN
400	0.241	1.7090	NAN
500	0.387	3.0840	NAN
600	0.569	5.6390	NAN
700	0.787	9.0150	NAN
800	1.058	13.7430	NAN
900	1.370	20.0860	NAN
1000	1.749	28.5770	NAN
1100	2.166	40.7380	NAN
1200	2.609	54.4120	NAN
1300	3.126	71.1180	NAN
1400	3.688	90.8250	NAN
1500	4.284	114.6030	NAN
1600	4.976	142.5150	NAN
1700	5.716	175.8220	NAN
1800	6.485	214.3160	NAN
1900	7.330	257.2930	NAN
2000	8.279	306.8480	NAN

Table 3.5: A comparison of timings (with a time limit of 1800 seconds) for multiplying $1 + X_1 + X_2$ by its inverse and updating the terms based on the precision between 100 and 2000.

Precision	BPAS	SAGE	MAPLE
25			2.434
50			65.252
75			573.291
100	0.580	9.820	3066.104(Broken pipe)
125		30.476	
150		74.160	
200	4.592	361.645	NAN
300	16.173	NAN	NAN
400	40.578	NAN	NAN
500	82.647	NAN	NAN
600	149.638	NAN	NAN
700	250.315	NAN	NAN
800	386.936	NAN	NAN
900	567.054	NAN	NAN
1000	710.227		
1100	980.659		
1200	1341.075		

Table 3.6: A comparison of timings (with a time limit of 1800 seconds) for multiplying $1 + X_1 + X_2 + X_3$ by its inverse and updating the terms based on the precision between 100 and 1200.

Chapter 4

The Design and Implementation of Univariate Polynomials over 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 coefficients, we add reference counting a UPOPS. The UPOPS struct can be seen in Listing 4.1.

```
typedef struct UPOPS {
    int deg;
    int alloc;
    PowerSeries_t** data;
    PowerSeries_t** weierstrassFData; //see Chapter 5
    int fDataSize;
    int refcount;
} UPOPS_t;
```

Listing 4.1: The univariate polynomial over power series struct.

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. [33, 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, multiplication, and negation follow easily. For completeness, we present simple algorithms for addition and multiplication of UPOPS in Algorithm 1 and Algorithm 2, respectively.

4.1 UPOPS Taylor Shift

One important operation on univariate polynomials over power series, which is not inherited directly from our power series implementation is Taylor shift. This operation takes a UPOPS

Algorithm 1 AddUnivariatePolynomialOverPowerSeries(f, g)

Input: $f = \sum_{i=0}^{k} a_i Y^i$, $g = \sum_{i=0}^{\ell} b_i Y^i$, $a_i, b_i \in \mathbb{K}[[X_1, \ldots, X_n]]$
Output: $h = \sum_{i=0}^{\max(k,\ell)} c_i Y^i = f + g$
1: $d = \min k, \ell$
2: **for** $i = 0$ to d **do** $c_i = a_i + b_i$
3: **for** $i = d + 1$ to k **do** $c_i = a_i$
4: **for** $i = d + 1$ to ℓ **do** $c_i = b_i$
5: **return** $h = \sum_{i}^{\max(k,\ell)} c_i Y^i$

Algorithm 2 MultiplyUnivariatePolynomialOverPowerSeries(f, g)

Input: $f = \sum_{i=0}^{k} a_i Y^i$, $g = \sum_{i=0}^{\ell} b_i Y^i$, $a_i, b_i \in \mathbb{K}[[X_1, \ldots, X_n]]$
Output: $h = \sum_{i=0}^{k+\ell} c_i Y^i = fg$
1: **for** $i = 0$ to $k + \ell$ **do** $c_i = 0$
2: **for** $i = 0$ to k **do**
3: | **for** $j = 0$ to ℓ **do**
4: | | $c_{i+j} = c_{i+j} + a_i \times b_j$
5: **return** $h = \sum_{i}^{k+\ell} c_i Y^i$

$f \in \mathbb{K}[[X_1, \ldots, X_n]][Y]$ and returns $f(Y + c)$ for some constant $c \in \mathbb{K}$. 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 Chapter 6), requires only shifting by elements of $\mathbb{K}$, and we thus specialize to that case.

Let us begin with a description of the particular problem. Given $f = \sum_{i=0}^{k} a_i Y^i$ we want to obtain $f(Y + c) = \sum_{i=0}^{k} a_i (Y + c)^i$. Since the coefficients of f are lazy power series, our goal is to compute $f(Y + c)$ lazily as well. That is, to compute $f(Y + c)$ in a way which relies on the underlying lazy power series arithmetic to yield a lazily computed UPOPS. Since our UPOPS are represented in a dense fashion, we compute the coefficients of $f(Y + c)$ as a polynomial in Y:

$$\begin{aligned} f(Y + c) = \; & a_0 + a_1(Y + c) + a_2(Y + c)^2 + a_3(Y + c)^3 + \ldots \\ = \; & (a_0 + ca_1 + c^2 a_2 + c^3 a_3 + \ldots) \\ & +(a_1 + 2ca_2 + 3c^2 a_3 + \ldots)Y \\ & +(a_2 + 3ca_3 + \ldots)Y^2 \\ & +(a_3 + \ldots)Y^3 + \ldots \end{aligned}$$

The coefficients of the expansion of $f(Y+c)$ create a triangular shape of linear combinations of the original coefficients of f. These linear combinations arise from the binomial expansion of $(Y + c)^i$ and are closely related to the Pascal triangle. Let $\mathbf{S}$ be a triangular matrix encoding the coefficients of the binomial expansion $(Y + c)^i$, for $i = 0, \ldots, k$, the vector $\mathbf{A}$ be the coefficients

of f, and the vector $\mathbf{B}$ be the coefficients of $f(Y+c)$. With $k=3$ we have:

$$\mathbf{S} = \begin{bmatrix} 1 & & & \\ 1 & c & & \\ 1 & 2c & c^2 & \\ 1 & 3c & 3c^2 & c^3 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Then we can verify that b_i as the inner product of the i-th sub-diagonal of $\mathbf{S}$ with the lower $k+1-i$ elements of $\mathbf{A}$, where k is the degree of f, 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 $j \leq i$. The homogeneous part of degree d, $b_{i_{(d)}}$, can be computed from only $a_{j_{(d)}}$, for $j \leq i$. Therefore, a generator for b_i is easily constructed from the homogeneous parts of a_j, for $j \leq i$, using multiplication by elements of $\mathbb{K}$ and polynomial addition. Therefore, in precisely this manner, 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, see Algorithm 3. That same algorithm is presented as in our actual C code as Listing 4.2. Finally, updating the coefficients of $f(Y+c)$ can be performed automatically, just as in any power series, by calling its generator via `homogPart_PS` or `polynomialPart_PS` (see Section 3.1). Since the main application of Taylor shift is factorization via Hensel's lemma, we leave its evaluation to Chapter 6 where benchmarks for factorization are presented.

Algorithm 3 LinearCombinationGenerator(d, f, $\mathbf{S}$, i)

Input: $f = \sum_{j=0}^{k} a_j Y^j$, $\mathbf{S} \in \mathbb{K}^{k+1 \times k+1}$ a lower triangular matrix of coefficients of $(Y+c)^j$ for $j = 0, \ldots, k$, the index i of the coefficient b_i for which to generate, d the requested degree

Output: $b_{i_{(d)}}$, the homogeneous part of degree d of b_i.

1: $b_{i_{(d)}} := 0$
2: **for** $\ell = i$ to k **do**
3: $\quad j := \ell + 1 - i$
4: $\quad b_{i_{(d)}} := b_{i_{(d)}} + S_{\ell+1,j} \times$ `homogPart_PS`(d, a_ℓ)
5: **return** $b_{i_{(d)}}$

```
Poly_ptr linearCombGen_PS(int d, long long i, mpq_t* S, Upops_t*  A) {
    int k = A->deg + 1;
    Poly_ptr ret = zeroPolynomial();
    for (int l = i; l < k; ++l) {
        int j = l - i;
        Poly_ptr homog = multiplyByRational(homogPart_PS(d, A->data[l]), S[l*k+j]);
        ret = addPolynomials(ret, homog);
    }
    return ret;
}

Poly_ptr linearCombGenVoid_PS(int d, void* p1, void* p2, void* p3) {
    return linearCombGen_PS(d, (long long) p1, (mpq_t*) p2, (Upops_t*) p3);
}

Upopt_t* Taylor_shift_initialization(mpq_t c_r, Upops_t* f) {
    mpq_t* S = getPascalTriLowerMat(c_r, k);
    Upops_t* B = allocateUnivariatePolynomialOverPowerSeries_UPOPS(f->deg+1);

    for (int row = 0; m <= f->deg; ++row) {
        B->data[row] = allocatePowerSeries_PS(1);
        B->data[row]->polys[0] = linearCombGen_PS(0, row, S, f);
        B->data[row]->deg = 0;

        reserve_UPOPS(f);
        B->data[row]->genOrder = 3;
        B->data[row]->genParam1 = (void*) n;
        B->data[row]->genParam2 = (void*) S;
        B->data[row]->genParam3 = (void*) f;
        B->data[row]->paramType1 = PLAIN_DATA;
        B->data[row]->paramType2 = PLAIN_DATA;
        B->data[row]->paramType3 = UPOPS;
        B->data[row]->gen.tertiaryGen = &(linearCombGenVoid_PS);
    }

    B->deg = f->deg;
    return B;
}
```

Listing 4.2: Computing the Taylor shift of a UPOPS lazily using a linear combination generator over the power series coefficients of the original UPOPS.

Chapter 5

A Lazy Weierstrass Preparation

In this chapter 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 2.2.1), we have that $f = \alpha p$ implies the following equalities:

$$\begin{array}{rcl} a_0 & = & b_0 c_0 \\ a_1 & = & b_0 c_1 + b_1 c_0 \\ & \vdots & \\ a_{d-1} & = & b_0 c_{d-1} + b_1 c_{d-2} + \cdots + b_{d-2} c_1 + b_{d-1} c_0 \\ a_d & = & b_0 c_d + b_1 c_{d-1} + \cdots + b_{d-1} c_1 + c_0 \\ & \vdots & \\ a_{d+m-1} & = & b_{d-1} c_m + c_{m-1} \\ a_{d+m} & = & c_m \end{array} \tag{5.1}$$

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. A simple rearrangement of the equations a_0 to a_{d-1} shows their successive dependency where first solving for b_{i-1} where it appears on the right hand side of an equation allows the next equation to be solved for b_i.

$$\begin{array}{rcl} a_0 & = & b_0 c_0 \\ a_1 - b_0 c_1 & = & b_1 c_0 \\ a_2 - b_0 c_2 - b_1 c_1 & = & b_2 c_0 \\ & \vdots & \\ a_{d-1} - b_0 c_{d-1} - b_1 c_{d-2} + \cdots - b_{d-2} c_1 & = & b_{d-1} c_0 \end{array} \tag{5.2}$$

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 $a_0 = b_0 c_0$ 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 (5.1) shows that each c_i may be computed modulo $\mathcal{M}^{r+1}$:

$$\begin{aligned} c_m &= a_{d+m} \\ c_{m-1} &= a_{d+m-1} - b_{d-1}c_m \\ c_{m-2} &= a_{d+m-2} - b_{d-2}c_m - b_{d-1}c_{m-1} \\ &\vdots \\ c_0 &= a_d - b_0c_d - b_1c_{d-1} - \cdots - b_{d-1}c_1 \end{aligned} \tag{5.3}$$

Consider the second equation. a_{d+m-1} and b_{d-1} are known modulo $\mathcal{M}^{r+1}$ and $b_{d-1} \in \mathcal{M}$ by definition. Then, the product $b_{d-1}c_m$ is known modulo $\mathcal{M}^{r+1}$ and we easily find 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 have been 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. Let us now see how we can successfully create generators for this purpose.

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 from being known modulo $\mathcal{M}^r$ to modulo $\mathcal{M}^{r+1}$ simultaneously. This so-called *Weierstrass update* follows two phases as just explained.

In the first phase of Weierstrass update, 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 $\mathcal{M}^{r+1}$:

$$\begin{aligned} f_{(1)} + f_{(2)} + \cdots + f_{(r)} &= (g_{(1)} + g_{(2)} + \cdots + g_{(r)})(h_{(0)} + h_{(1)} + \cdots + h_{(r)}) \\ &= \left(g_{(1)}h_{(0)}\right) + \left(g_{(2)}h_{(0)} + g_{(1)}h_{(1)}\right) + \cdots + \\ &\quad \left(g_{(r)}h_{(0)} + g_{(r-1)}h_{(1)} + \cdots + g_{(1)}h_{(r-1)}\right). \end{aligned} \tag{5.4}$$

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

$$f_{(r)} = g_{(r)}h_{(0)} + g_{(r-1)}h_{(1)} + \cdots + g_{(1)}h_{(r-1)}, \tag{5.5}$$

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.

Algorithm 4 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 2.2.1, $\mathcal{F} = \{F_i \mid F_i = a_i - \sum_{k=0}^{i} b_k c_{i-k}, i = 0, \ldots, d-1\}$, 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) × 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.
```

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 (5.2). 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 (5.5), 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 (5.3) 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 4, 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 (see Listing 4.1).

Notice 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 and could be performed with a parallel map. Structuring Weierstrass preparation as a lazy operation also naturally exposes further concurrency opportunities, as in the case of factorization via Hensel's lemma, see Chapter 6.

5.1 Benchmarks: Weierstrass Preparation

In this section, we report on the experimental data of the implementation of Weierstrass preparation in BPAS against that of MAPLE's `PowerSeries` library. We have studied two families of examples for $d \geq 3$:

$$\frac{1}{1 + X_1 + X_2} Y^d + X_2 Y^{d-1} + \ldots + Y + X_1$$

$$\frac{1}{1 + X_1 + X_2} Y^d + Y^{d-1} + \ldots + X_2 Y + X_1$$

The results are shown in Tables 5.1- 5.8 and Figures 5.1 and 5.2 gather data from these Tables in a graphical manner. 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 the efficient underlying power series arithmetic we have implemented, as well as our smart implementation of Lemma 1. Consider the following example

$$\frac{1}{1 + X_1 + X_2} Y^3 + Y^2 + X_2 Y + X_1$$

Applying Weierstrass Preparation modulo terms of degree 4 gives the following

$$\begin{aligned} p &= X_1 + X_2X_1 - X_1^2 + X_2^2X_1 - 4X_2X_1^2 + 5X_1^3 \\ &\quad +(X_2 - X_1 + X_2^2 - 2X_1X_2 + 3X_1^2 + X_2^3 - 6X_2^2X_1 + 13X_2X_1^2 - 14X_1^3)Y + Y^2 \\ \alpha &= 1 - X_2 + X_1 + 2X_1X_2 - 4X_1^2 - X_2^3 + 4X_2^2X_1 - 11X_2X_1^2 + 18X_1^3 \\ &\quad +(1 - X_2 - X_1 + X_2^2 + 2X_1X_2 + X_1^2 - X_2^3 - 3X_2^2X_1 - 3X_2X_1^2 - X_1^3)Y \end{aligned}$$

Precision	BPAS	MAPLE
10	0.002	0.247
20	0.020	3.279
30	0.073	20.315
40	0.200	85.156
50	0.442	210.591
60	0.863	541.932
70	1.526	1024.378
80	2.460	NAN
90	3.708	NAN
100	5.415	NAN
110	7.715	NAN
120	10.738	NAN
130	14.631	NAN
140	19.551	NAN
150	25.665	NAN
160	33.132	NAN
170	42.719	NAN
180	53.362	NAN
190	66.691	NAN
200	82.007	NAN
210	100.508	NAN
220	123.160	NAN
230	146.332	NAN
240	175.616	NAN
250	208.458	NAN
260	245.851	NAN
270	288.241	NAN
280	339.781	NAN
290	394.882	NAN
300	456.991	NAN
310	523.884	NAN
320	601.067	NAN
330	692.865	NAN
340	783.561	NAN
350	889.407	NAN
360	1006.675	NAN
370	1136.410	NAN
380	1280.674	NAN
390	1448.138	
400	1622.192	

Table 5.1: A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^3 + Y^2 + X_2Y + X_1$ to a precision between 10 and 400.

Precision	BPAS	MAPLE
10	0.002	0.273
20	0.021	3.804
30	0.097	24.839
40	0.290	79.437
50	0.668	229.678
60	1.321	539.220
70	2.333	1132.865
80	3.847	NAN
90	5.985	NAN
100	8.969	NAN
110	12.891	NAN
120	18.044	NAN
130	25.172	NAN
140	33.356	NAN
150	43.436	NAN
160	56.074	NAN
170	71.667	NAN
180	90.146	NAN
190	112.311	NAN
200	139.663	NAN
210	168.948	NAN
220	204.669	NAN
230	248.443	NAN
240	294.819	NAN
250	349.257	NAN
260	412.409	NAN
270	483.739	NAN
280	565.858	NAN
290	656.939	NAN
300	763.749	NAN
310	873.289	NAN
320	1005.939	NAN
330	1146.149	NAN
340	1304.197	NAN
350	1482.199	NAN
360	broken pipe	NAN
370	broken pipe	
380	broken pipe	
390	broken pipe	
400	broken pipe	

Table 5.2: A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^4 + Y^3 + Y^2 + X_2Y + X_1$ to a precision between 10 and 400.

Precision	BPAS	MAPLE
10	0.004	0.304
20	0.033	4.651
30	0.133	24.930
40	0.394	94.599
50	0.898	249.586
60	1.762	567.922
70	2.947	1152.821
80	4.682	NAN
90	7.124	NAN
100	10.502	NAN
110	14.973	NAN
120	20.864	NAN
130	28.498	NAN
140	38.012	NAN
150	50.166	NAN
160	64.837	NAN
170	83.171	NAN
180	104.395	NAN
190	129.709	NAN
200	160.147	NAN
210	195.474	NAN
220	237.051	NAN
230	285.917	NAN
240	343.213	NAN
250	405.442	NAN
260	478.299	NAN
270	562.075	NAN
280	657.658	NAN
290	764.034	NAN
300	885.143	NAN
310	1021.389	NAN
320	1182.772	NAN
330	1333.249	NAN
340	1531.011	NAN
350	1744.344	NAN
360	broken pipe	
370	broken pipe	
380	broken pipe	
390	broken pipe	
400	broken pipe	

Table 5.3: A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^5 + Y^4 + Y^3 + Y^2 + X_2Y + X_1$ to a precision between 10 and 400.

Precision	BPAS	MAPLE
10	0.003	0.325
20	0.033	4.371
30	0.153	30.760
40	0.475	88.231
50	1.110	251.065
60	2.164	600.943
70	3.649	1212.061
80	5.853	NAN
90	9.108	NAN
100	13.103	NAN
110	18.964	NAN
120	26.314	NAN
130	35.949	NAN
140	48.637	NAN
150	62.361	NAN
160	80.877	NAN
170	103.936	NAN
180	135.512	NAN
190	160.955	NAN
200	198.786	NAN
210	242.428	NAN
220	306.591	NAN
230	355.826	NAN
240	423.311	NAN
250	504.435	NAN
260	594.056	NAN
270	696.032	NAN
280	811.279	NAN
290	941.532	NAN
300	1098.941	NAN
310	1258.003	NAN
320	1446.503	NAN
330	1651.794	NAN
340	broken pipe	
350	broken pipe	
360	broken pipe	
370	broken pipe	
380	broken pipe	
390	broken pipe	
400	broken pipe	

Table 5.4: A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^6 + Y^5 + Y^4 + Y^3 + Y^2 + X_2Y + X_1$ to a precision between 10 and 400.

Precision	BPAS	MAPLE
10	0.001	0.151
20	0.011	1.729
30	0.037	10.031
40	0.105	37.609
50	0.253	105.318
60	0.536	258.575
70	0.986	539.848
80	1.711	998.493
90	2.634	1780.272
100	3.862	NAN
110	5.496	NAN
120	7.602	NAN
130	10.404	NAN
140	13.824	NAN
150	18.093	NAN
160	22.945	NAN
170	29.113	NAN
180	36.442	NAN
190	45.028	NAN
200	55.040	NAN
210	66.827	NAN
220	80.244	NAN
230	96.138	NAN
240	113.384	NAN
250	133.796	NAN
260	156.909	NAN
270	181.917	NAN
280	211.414	NAN
290	244.005	NAN
300	280.072	NAN
310	319.831	NAN
320	367.431	NAN
330	414.084	NAN
340	469.168	NAN
350	528.372	NAN
360	593.875	NAN
370	666.165	NAN
380	742.869	NAN
390	832.976	NAN
400	920.777	NAN
410	1020.612	NAN
420	1129.975	NAN
430	1247.622	NAN
440	1379.826	NAN
450	1512.082	NAN
460	1664.595	NAN

Table 5.5: A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^3 + X_2Y^2 + Y + X_1$ to a precision between 10 and 460.

Precision	BPAS	MAPLE
10	0.001	0.161
20	0.012	1.757
30	0.045	10.645
40	0.138	40.499
50	0.342	108.484
60	0.720	258.809
70	1.351	539.721
80	2.290	1021.864
90	3.605	1803.078
100	5.414	NAN
110	7.719	NAN
120	10.842	NAN
130	14.886	NAN
140	19.964	NAN
150	26.209	NAN
160	33.974	NAN
170	42.916	NAN
180	53.829	NAN
190	66.768	NAN
200	81.864	NAN
210	99.576	NAN
220	119.933	NAN
230	145.626	NAN
240	169.923	NAN
250	200.974	NAN
260	235.357	NAN
270	275.140	NAN
280	319.773	NAN
290	367.182	NAN
300	426.221	NAN
310	489.159	NAN
320	552.589	NAN
330	625.719	NAN
340	712.574	NAN
350	806.494	NAN
360	906.838	NAN
370	1017.376	NAN
380	1131.829	NAN
390	1262.769	NAN
400	1409.174	NAN
410	1576.995	NAN
420	1724.758	NAN

Table 5.6: A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^4 + X_2Y^3 + Y^2 + Y + X_1$ to a precision between 10 and 420.

Precision	BPAS	MAPLE
10	0.001	0.166
20	0.011	1.795
30	0.045	9.983
40	0.145	37.168
50	0.369	114.697
60	0.799	270.756
70	1.543	566.299
80	2.646	1071.545
90	4.197	1895.376(broken pipe)
100	6.372	NAN
110	9.320	NAN
120	13.292	NAN
130	18.088	NAN
140	24.403	NAN
150	32.124	NAN
160	41.091	NAN
170	52.217	NAN
180	65.604	NAN
190	81.333	NAN
200	99.469	NAN
210	120.958	NAN
220	145.547	NAN
230	175.619	NAN
240	206.118	NAN
250	242.600	NAN
260	284.389	NAN
270	331.255	NAN
280	382.335	NAN
290	442.263	NAN
300	511.319	NAN
310	580.384	NAN
320	665.154	NAN
330	747.303	NAN
340	844.106	NAN
350	955.535	NAN
360	1071.794	NAN
370	1205.771	NAN
380	1352.387	NAN
390	1500.987	NAN
400	1669.323	NAN

Table 5.7: A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^5 + X_2Y^4 + Y^3 + Y^2 + Y + X_1$ to a precision between 10 and 400.

Precision	BPAS	MAPLE
10	0.001	0.172
20	0.007	2.431
30	0.045	12.726
40	0.139	50.591
50	0.357	129.982
60	0.790	271.258
70	1.335	572.419
80	2.471	1078.278
90	4.179	1930.709(Broken pipe)
100	6.330	NAN
110	9.300	NAN
120	13.148	NAN
130	18.101	NAN
140	24.315	NAN
150	31.992	NAN
160	41.338	NAN
170	53.367	NAN
180	66.056	NAN
190	82.582	NAN
200	101.408	NAN
210	121.978	NAN
220	147.209	NAN
230	175.750	NAN
240	208.157	NAN
250	245.306	NAN
260	287.679	NAN
270	335.511	NAN
280	388.080	NAN
290	453.190	NAN
300	514.074	NAN
310	587.441	NAN
320	669.546	NAN
330	761.345	NAN
340	860.185	NAN
350	968.260	NAN
360	1089.307	NAN
370	1221.999	NAN
380	1363.928	NAN
390	1536.889	NAN
400	1687.272	NAN

Table 5.8: A comparison of timings (with a time limit of 1800 seconds) for Weierstrass preparation results for $\frac{1}{1+X_1+X_2}Y^6 + X_2Y^5 + Y^4 + Y^3 + Y^2 + Y + X_1$ to a precision between 10 and 400.

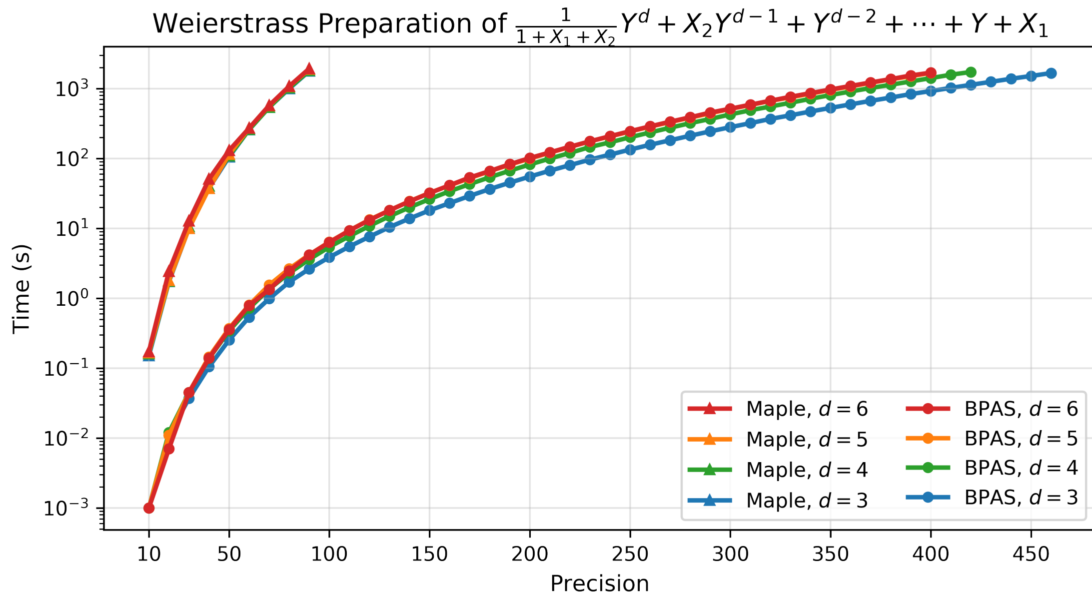

Figure 5.1: Applying Weierstrass preparation on family (i) for increasing precisions.

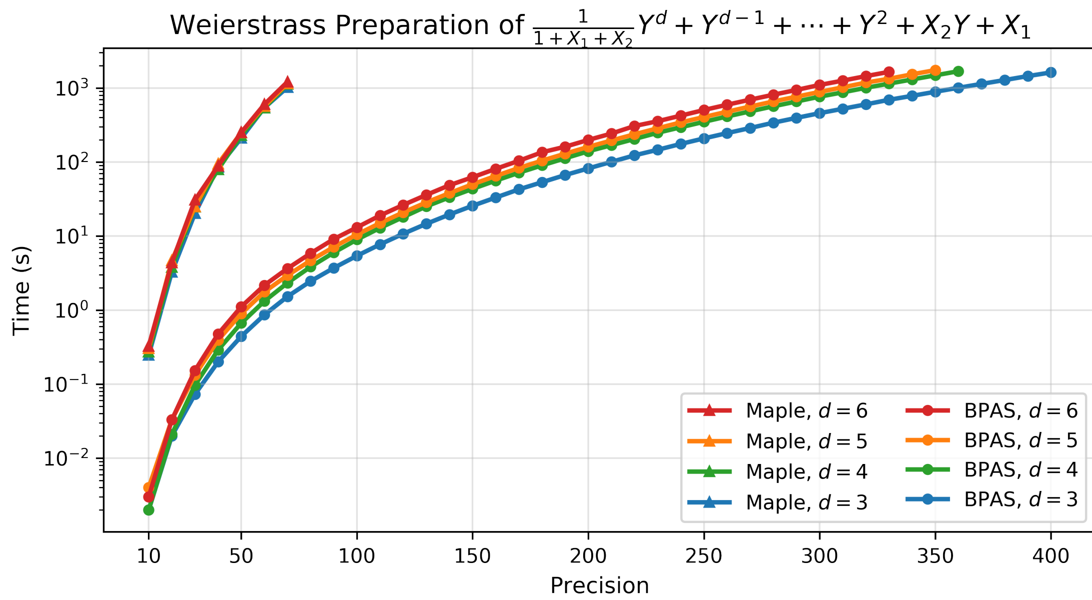

Figure 5.2: Applying Weierstrass preparation on family (ii) for increasing precisions.

Chapter 6

A Lazy Factorization via Hensel's Lemma

In Section 2.3 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. In this chapter we look to obtain a lazy implementation of this construction.

Recall that the proof of Theorem 2.3.1 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 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 is general in Y of order k_i. 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 the UPOPS to factorize in the next step as $f^* = \alpha(Y - c_r)$. The full procedure for obtaining all factors of f is shown as an iterative process, instead of recursive, in Algorithm 5.

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 4.1) and Weierstrass preparation (Chapter 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 Taylor shift operations on top of a Weierstrass update.

Notice, however, that the order in which factors are created matters. Indeed, the first factor to be created can be updated independent of the rest, since it is the result of a single Weierstrass preparation. However, later factors are created by applying Weierstrass preparation on some α, the result of a previous Weierstrass preparation. Therefore, the ancestry of later factors is much deeper, and a cascade of updates through several different Weierstrass updates are required to update these later factors with deep hierarchies. A useful optimization which could then be made is to order the creation of the factors so that more important ones are created first.

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

Algorithm 5 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 2.3.1.

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** $f_1, \ldots, f_r$

preparation. They in essence form a *pipeline* of processes [23, 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.

6.1 Benchmarks: Factorization via Hensel's Lemma

In this section we 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 5 while EHC factorizes UPOPS over the field of Puiseux series in $X_1, \ldots, X_n$, see [2]. 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.

Tables 6.1 and 6.2 show the results of this experimentation. As an example, consider the first UPOPS:

$$f = (Z - 1)(Z - 2)(Z - 3) + X_1(Z^2 + Z).$$

Applying the Hensel factorization modulo terms of degree 5 results in the following branches

$$\begin{aligned} Z &= 1 - X_1 + 3X_1^2 - \frac{27}{2}X_1^3 + \frac{291}{4}X_1^4 \\ Z &= 2 + 6X_1 + 30X_1^2 + 402X_1^3 + 5610X_1^4 \\ Z &= 3 - 6X_1 - 33X_1^2 - \frac{777}{2}X_1^3 - \frac{22731}{4}X_1^4 \end{aligned}$$

Figure 6.1 gathers data from Tables 6.1 and 6.2 in a graphical manner 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 5 is work in progress.

Precision	BPAS	EHC	FVHL
25			4.541
50			38.989
100	0.029	1.596	450.451
150			LostConnectionKernel
200	0.106	7.836	NAN
300	0.264	23.359	NAN
400	0.516	52.362	NAN
500	0.891	100.546	NAN
600	1.420	175.084	NAN
700	2.064	280.707	NAN
800	2.817	430.611	NAN
900	3.778	636.562	NAN
1000	5.000	898.689	NAN
1100	6.430	1249.207	NAN
1200	8.259	1750.483	NAN
1300	10.253	NAN	NAN
1400	12.697	NAN	NAN
1500	15.519	NAN	NAN
1600	18.951	NAN	NAN
1700	22.698	NAN	NAN
1800	26.908	NAN	NAN
1900	31.866	NAN	NAN
2000	37.372	NAN	NAN

Table 6.1: Factorization via Hensel for $(Z-1)(Z-2)(Z-3)+X_1(Z^2+Z)$ to a precision between 25 and 2000. Here, EHC and FVHL stand for the `ExtendedHenselConstruction` and `FactorizationViaHenselLemma` functions, respectively. Note that the benchmarks were collected with a time limit of 1800 seconds.

Precision	BPAS	EHC	FVHL
25			7.525
50			56.198
100	0.101	22.991	662.591
150			LostConnectionToKernel
200	0.849	339.767	NAN
300	2.987	1668.969	NAN
400	7.312	NAN	NAN
500	14.702	NAN	NAN
600	26.057	NAN	NAN
700	42.447	NAN	NAN
800	64.853	NAN	NAN
900	94.576	NAN	NAN
1000	132.263	NAN	NAN
1100	179.757	NAN	NAN
1200	238.469	NAN	NAN
1300	309.627	NAN	NAN
1400	393.539	NAN	NAN
1500	493.704	NAN	NAN
1600	608.882	NAN	NAN
1700	703.081		
1800	844.859		
1900	1006.340		
2000	1189.676		

Table 6.2: Factorization via Hensel for $(Z-1)(Z-2)(Z-3)(Z-4) + X_1(Z^3+Z)$ to a precision between 25 and 2000. Here, EHC and FVHL stand for the `ExtendedHenselConstruction` and `FactorizationViaHenselLemma` functions, respectively. Note that the benchmarks were collected with a time limit of 1800 seconds.

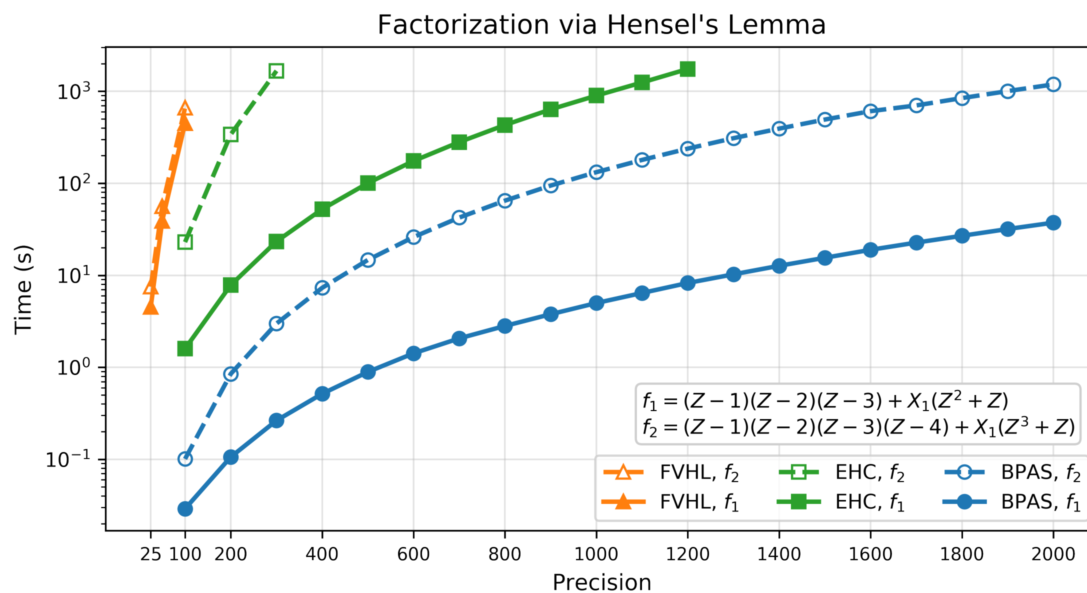

Figure 6.1: Applying factorization via Hensel's lemma to the UPOPS $f_1 = (Z-1)(Z-2)(Z-3) + X_1(Z^2+Z)$ and $f_2 = (Z-1)(Z-2)(Z-3)(Z-4) + X_1(Z^3+Z)$.

Chapter 7

Applications in Bifurcation Theory

7.1 Introduction

Consider the smooth map

$$\Phi : \mathbb{R}^n \times \mathbb{R}^m \longrightarrow \mathbb{R}^n, \quad \Phi_i(\mathbf{x}, \boldsymbol{\alpha}) = 0, \; i = 0, \ldots, n \tag{7.1}$$

where the vectors $\mathbf{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_m)$ represent state variables and parameters, respectively. We assume that $\Phi_i(\mathbf{0}, \mathbf{0}) = 0$. The smooth map Φ is called *singular* when $\det(d\Phi)_{(\mathbf{0},\mathbf{0})} = 0$. The local zeros of a singular map may experience *qualitative* changes when small perturbations are applied to the parameters $\boldsymbol{\alpha}$. These changes are called *bifurcation*. Local bifurcation analysis of zeros of the singular smooth map (7.1) plays a pivotal role in exploring the behaviour of many real world problems [14–16, 21]. *Lyapunov-Schmidt reduction* is a fundamental tool converting the singular map (7.1) into $g : \mathbb{R}^p \times \mathbb{R}^m \longrightarrow \mathbb{R}^p$ with $p = n - \operatorname{rank}(d\Phi_{\mathbf{0},\mathbf{0}})$. The reduction is achieved through producing an equivalent map to (7.1) made up of a pair of equations and making use of the Implicit Function Theorem. This solves the $n-1$ variables of $\mathbf{x}$ in the first equation; thereafter, substituting the result into the second one gives an equation for the remaining variable. It is proved that the local zeros of the map g are in one-to-one correspondence with the local zeros of Φ; for more details see [16, Pages 25–34]. Hence, the study of local zeros of (7.1) is facilitated through treating their counterparts in g. Singularity theory is an approach providing a comprehensive framework equipped with effective tools for this study. The pioneering work of René Thom established the original ideas of the theory which was then extensively developed by John Mather and V. I. Arnold. The book series [16] written by Marty Golubitsky, Ian Stewart and David G. Schaeffer is a collection of significant contributions of the authors in dealing with a wide range of real world problems using singularity theory techniques as well as explaining the underlying ideas of the theory in ways accessible to applied scientists and mathematicians particularly those dealing with bifurcation problems in the presence of parameters and symmetries. The singularity theory tools are applied to the problems that have emerged as an output of the Lyapunov-Schmidt reduction. Following [16, Page 25], we focus on the reduction when $\operatorname{rank}(d\Phi_{\mathbf{0},\mathbf{0}}) = n-1$, $m = 1$ and refer to $\alpha_1 = \lambda$ as the bifurcation parameter. In other words, we consider the following map

$$g : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R} \quad g(x, \lambda) = 0. \tag{7.2}$$

Two smooth maps are regarded as *germ-equivalent* when they are identical on some neighborhood of the origin. In fact, a *germ-equivalence* class of a smooth map is called a *germ*. We denote by $\mathscr{E}_{x,\lambda}$ the space of all scalar smooth germs which is a local ring with $\mathcal{M}_{\mathscr{E}_{x,\lambda}} = \langle x, \lambda \rangle_{\mathscr{E}_{x,\lambda}}$ as the unique maximal ideal; see also [16, Page 56] and [14, Page 3]. Due to the existence of germs with infinite Taylor series and flat germs (whose Taylor series is zero), there does not exist a computational tool to automatically study local bifurcations in $\mathscr{E}_{x,\lambda}$. This has motivated the authors of [14] to propose circumstances under which the computations supporting the bifurcation analysis in $\mathscr{E}_{x,\lambda}$ are transferred to smaller local rings and verify that the corresponding results are valid in $\mathscr{E}_{x,\lambda}$. For instance, the following theorem permits the use of formal power series $K[[x, \lambda]]$ ring as a smaller computational ring in computation of algebraic objects involved in the analysis of bifurcation.

Theorem 7.1.1 *([14, Theorem 4.3]) Suppose that $\{f_i\}_{i=1}^m \in \mathscr{E}_{x,\lambda}$. For $k, N \in \mathbb{N}$ with $k \leq N$,*

$$\mathcal{M}^k_{K[[x,\lambda]]} \subseteq \langle J^N f_1, \ldots, J^N f_m \rangle_{K[[x,\lambda]]} \quad \textit{iff} \quad \mathcal{M}^k_{\mathscr{E}_{x,\lambda}} \subseteq \langle f_1, \ldots, f_m \rangle_{\mathscr{E}_{x,\lambda}}$$

where $\mathcal{M}^k = \langle x^{\alpha_1} \lambda^{\alpha_2} : \alpha_1 + \alpha_2 = k \rangle$ and $J^N f_i$ is the sum of terms of degree N or less in the Taylor series of f_i.

This, along with other criteria in [13, 14], highlights the importance of alternative rings in performing automatic local bifurcation analysis of scalar and $\mathbb{Z}_2$-equivariant singularities.

The work presented here addresses one of the applications of the so-called Extended Hensel Construction (EHC) invented by Sasaki and Kako, see [28]. We show that the EHC can be used in computing the reduced system $g \in \mathscr{E}_{x,\lambda}$, which, as a result, leads to determining the type of singularity hidden in system (7.1).

This EHC has been studied and improved by many authors. In particular, the papers [2–4] present algorithmic improvements (where the EHC relies only linear algebra techniques and univariate polynomial arithmetic) together with applications of the EHC in deriving real branches of space curves and consequently computing limits of real multivariate rational functions. The same authors implemented their version of the EHC as the `ExtendedHenselConstruction` command of the `PowerSeries` library [1].

The EHC comes into two flavors. In the case of bivariate polynomials it behaves as Newton-Puiseux algorithm while with multivariate polynomials it acts as an effective version of Jung-Abhyankar Theorem. In both cases, it provides a factorization of the input object in the vicinity of the origin. We believe that this capability makes the EHC a desirable tool for an automatic derivation of the zeros of a polynomial system locally near the origin. The rest of this paper is organized as follows. In Section 7.2, some of the ideas in singularity theory are reviewed. We then discuss the EHC procedure followed by an overview on the `PowerSeries` Library. Finally, our proposed approach is illustrated through two examples revealing pitchfork and winged cusp bifurcations.

[1] `http://www.regularchains.org/downloads.html`

7.2 Background

7.2.1 Concepts from Singularity Theory

In this section we explain the materials required for defining recognition problem of a singular germ. These concepts are accompanied by examples. We skip the technical details of singularity theory-related concepts as they are beyond the scope of this paper. The interested readers are referred to [12, 14, 16] for the principal ideas, algebraic formulations and automatic computation of the following objects.

Contact equivalence. We say that two smooth germs $f, g \in \mathscr{E}_{x,\lambda}$ are *contact equivalent* when

$$g(x, \lambda) = S(x, \lambda) f(X(x, \lambda), \Lambda(\lambda)) \tag{7.3}$$

is held for a smooth germ $S(x, \lambda) \in \mathscr{E}_{x,\lambda}$ and local diffeomorphisms $((x, \lambda) \longrightarrow (X(x, \lambda), \Lambda(\lambda))) : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$ satisfying

$$S(x, \lambda), X_x(x, \lambda), \Lambda(\lambda) > 0$$

Normal form. Bifurcation analysis of local zeros of g in (7.2) requires computing a contact equivalent germ to g which has simpler structure and makes the analysis efficient. Indeed, each step of this analysis, for instance recognition problem, involves normal form computation. To be more precise, the simplest representative of the class of $g \in \mathscr{E}_{x,\lambda}$ under contact equivalence is called a *normal form* of g.

Example Consider the smooth germ $g(x, \lambda) = \sin(x^3) - \lambda x + \exp(\lambda^3) - 1 \in \mathscr{E}_{x,\lambda}$. Note that $g(0, 0) = \frac{\partial}{\partial x} g(0, 0) = 0$; therefore, the origin is the singular point of g. The procedure in [14, Section 6] returns $x^3 - \lambda x$ as the normal form of g denoted by NF(g). The equation $x^3 - \lambda x = 0$ is called the *pitchfork bifurcation problem* and the *bifurcation diagram* for pitchfork is defined by the local variety $\{(x, \lambda) \mid x^3 - x\lambda = 0\}$. When λ smoothly varies around the origin, the number of solutions of the pitchfork bifurcation problem changes from one to three; see Figure 7.1.

Now, modulo monomials of degree ≥ 5, we compute the transformation $(X(x, \lambda), S(x, \lambda), \Lambda(\lambda))$ through which g is converted into NF(g) in (7.3).

$$\begin{aligned} X(x, \lambda) &:= x + \lambda^2 + \lambda x + \lambda^2 x + \lambda x^2 + x^3, \\ S(x, \lambda) &:= 1 - \lambda + 2\lambda x + x^2, \\ \Lambda(\lambda) &:= \lambda. \end{aligned}$$

Recognition problem. Let $g \in \mathscr{E}_{x,\lambda}$ be a singular germ. *Recognition problem* for a normal form of g computes a list of zero and non-zero conditions on derivatives of a singular germ $f \in \mathscr{E}_{x,\lambda}$ under which f is contact-equivalent to g. The proposed algorithm in [16, Pages 86–93], divides monomials (in $\mathscr{E}_{x,\lambda}$) into three categories; *low*, *intermediate* and *high order terms*. *Low order terms* refer to the monomials of the form $x^{\alpha_1}\lambda^{\alpha_2}$ that do not participate in the representation of any germ equivalent to g. The *high order terms* consist of the monomials $x^{\alpha_1}\lambda^{\alpha_2}$ which do not change the structure of the local zeros of g when they are present; that is, adding $x^{\alpha_1}\lambda^{\alpha_2}$ to g creates a germ contact equivalent to g. Due to the sophisticated structure of intermediate order terms we skip defining them here and instead introduce *intrinsic* generators

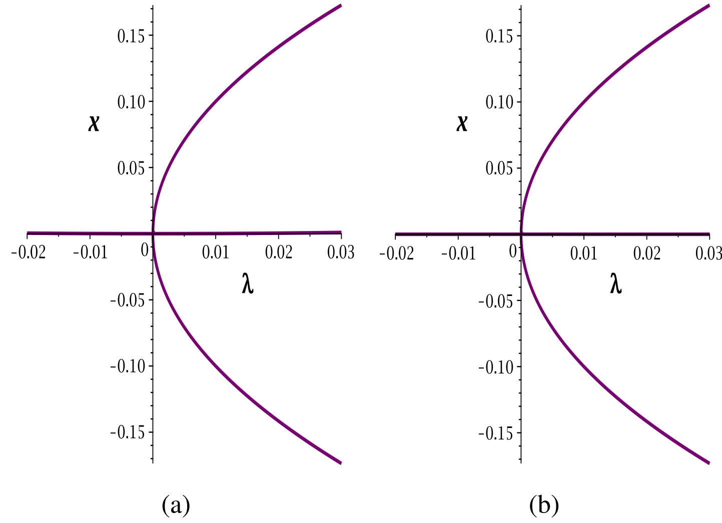

Figure 7.1: Figures (a) and (b) depict the bifurcation diagrams of g and $\mathrm{NF}(g)$, respectively.

$x^{\alpha_1}\lambda^{\alpha_2}$ which contribute to every equivalent germ and provide information about intermediate order terms. Low order terms and intrinsic generators are identified through the following theorem.

Theorem 7.2.1 *[16, Theorems 8.3 and 8.4, Page 88] Suppose that $f, g \in \mathscr{E}_{x,\lambda}$ and there exists a positive integer k such that $\mathcal{M}^k_{\mathscr{E}_{x,\lambda}} \subset \langle g, x\frac{\partial}{\partial x}g, \lambda\frac{\partial}{\partial x}g\rangle_{\mathscr{E}_{x,\lambda}}$.*

- (a) *if f is equivalent to g and $x^{\alpha_1}\lambda^{\alpha_2}$ belongs to low order terms of g then $\frac{\partial^{\alpha_1}}{\partial x^{\alpha_1}}\frac{\partial^{\alpha_2}}{\partial \lambda^{\alpha_2}}f(0,0) = 0$.*
- (b) *furthermore, assume that $x^{\alpha_1}\lambda^{\alpha_2}$ belongs to intrinsic generators of g. If f is equivalent to g then $\frac{\partial^{\alpha_1}}{\partial x^{\alpha_1}}\frac{\partial^{\alpha_2}}{\partial \alpha^{\alpha_2}}f(0,0) \neq 0$.*

Example For the smooth germ g given by Example 7.2.1, we deduce the vector space $\mathbb{R}\{1, \lambda, x, x^2\}$ as low order terms. It follows from Theorem 7.2.1(a) that any germ f equivalent to g satisfies

$$f(0,0) = \frac{\partial}{\partial \lambda}f(0,0) = \frac{\partial}{\partial x}f(0,0) = \frac{\partial^2}{\partial x^2}f(0,0) = 0 \tag{7.4}$$

Moreover, the higher order terms of g are determined by the ideal

$$\langle x^4, \lambda^4, x^3\lambda, x\lambda^3, x^2\lambda^2\rangle_{\mathscr{E}_{x,\lambda}} + \langle x^2\lambda, \lambda^3, x\lambda^2\rangle_{\mathscr{E}_{x,\lambda}} + \langle \lambda^2\rangle_{\mathscr{E}_{x,\lambda}}$$

which means that adding/removing any monomial, taken from this ideal, to/from g gives a new germ equivalent to g. Finally, the corresponding intrinsic generators of g are described via $\{x^3, \lambda x\}$ verified by Theorem 7.2.1(b) that for any germ f equivalent to g the following is valid

$$\frac{\partial^3}{\partial x^3}f(0,0) \neq 0, \quad \frac{\partial}{\partial \lambda}\frac{\partial}{\partial x}f(0,0) \neq 0 \tag{7.5}$$

To sum up, the recognition problem for a normal form of g is characterized by (7.4) and (7.5).

7.2.2 The Extended Hensel Construction

This part is summarized from [2].

Notation 1 Suppose that $\mathbb{K}$ is an algebraic number field whose algebraic closure is denoted by $\overline{\mathbb{K}}$. Assume that $F(X,Y) \in \mathbb{K}[X,Y]$ is a bivariate polynomial with complex number coefficients. Let also F be a univariate polynomial in X which is monic and square-free. The partial degree of F w.r.t. X is represented by d. We denote by $\mathbb{K}[[U^*]] = \bigcup_{\ell=1}^{\infty} \mathbb{K}[[U^{\frac{1}{\ell}}]]$ the ring of *formal Puiseux series*. Hence, given $\varphi \in \mathbb{K}[[U^*]]$, there exists $\ell \in \mathbb{N}_{>0}$ such that $\varphi \in \mathbb{K}[[U^{\frac{1}{\ell}}]]$ holds. Thus, we can write $\varphi = \sum_{m=0}^{\infty} a_m U^{\frac{m}{\ell}}$, for some $a_0, \ldots, a_m, \ldots \in \mathbb{K}$. We denote by $\mathbb{K}((U^*))$ the quotient field of $\mathbb{K}[[U^*]]$. Let $\varphi \in \mathbb{K}[[U^*]]$ and $\ell \in \mathbb{N}$ such that $\varphi = f(U^{\frac{1}{\ell}})$ holds for some $f \in \mathbb{K}[[U]]$. We say that the Puiseux series φ is *convergent* if we have $f \in \mathbb{K}\langle U\rangle$. We recall Puiseux's theorem: if $\mathbb{K}$ is an algebraically closed field of characteristic zero, the field $\mathbb{K}((U^*))$ of formal Puiseux series over $\mathbb{K}$ is the algebraic closure of the field of formal Laurent series over $\mathbb{K}$; moreover, if $\mathbb{K} = \mathbb{C}$, then the field $\mathbb{C}(\langle Y^*\rangle)$ of convergent Puiseux series over $\mathbb{C}$ is algebraically closed as well.

The purpose of the EHC is to factorize $F(X,Y)$ as $F(X,Y) = G_1(X,Y) \cdots G_r(X,Y)$, with $G_i(X,Y) \in \overline{\mathbb{K}}(\langle Y^*\rangle)[X]$ and $\deg_X(G_i) = m_i$, for $1 \le i \le r$. Thus, the EHC factorizes $F(X,Y)$ over $\overline{\mathbb{K}}(\langle Y^*\rangle)$, thus over $\mathbb{C}(\langle Y^*\rangle)$.

Newton line. We plot each non-zero term $c\,X^{e_x}Y^{e_y}$ of $F(X,Y)$ to the point of coordinates (e_x, e_y) in the Euclidean plane equipped with Cartesian coordinates. We call *Newton Line* the straight line L passing through the point $(d,0)$ and another point, such that no other points lie below L. The equation of L is $e_x/d + e_y/\delta = 1$ for some $\delta \in \mathbb{Q}$. We define $\widehat{\delta}, \widehat{d} \in \mathbb{Z}^{>0}$ such that $\widehat{\delta}/\widehat{d} = \delta/d$ and $\gcd(\widehat{\delta}, \widehat{d}) = 1$ both hold.

Newton polynomial. The sum of all the terms of $F(X,Y)$ which are plotted on the Newton line of F is called the *Newton polynomial* of F. We denote it by $F^{(0)}$. Observe that the Newton polynomial is a homogeneous polynomial in $(X, Y^{\delta/d})$. Let $\zeta_1, \ldots, \zeta_r \in \overline{\mathbb{K}}$ be the distinct roots of $F^{(0)}(X,1)$, for some $r \ge 2$. Hence we have $\zeta_i \ne \zeta_j$ for all $1 \le i < j \le r$ and there exist positive integers $m_1 \le m_2 \le \cdots \le m_r$ such that, using the homogeneity of $F^{(0)}(X,Y)$, we have

$$F^{(0)}(X,Y) = (X - \zeta_1 Y^{\delta/d})^{m_1} \cdots (X - \zeta_r Y^{\delta/d})^{m_r}.$$

The *initial factors* of $F^{(0)}(X,Y)$ are $G_i^{(0)}(X,Y) := (X - \zeta_i Y^{\delta/d})^{m_i}$, for $1 \le i \le r$. For simplicity, we put $\widehat{Y} = Y^{\widehat{\delta}/\widehat{d}}$.

Theorem 7.2.2 (Extended Hensel Construction) *We define the ideal*

$$S_k = \langle X^d Y^{(k+0)/\widehat{d}}, X^{d-1} Y^{(k+\widehat{\delta})/\widehat{d}}, \ldots, X^0 Y^{(k+d\widehat{\delta})/\widehat{d}} \rangle, \tag{7.6}$$

for $k = 1, 2, \ldots$. Then, for all integer $k > 0$, we can construct $G_i^{(k)}(X,Y) \in \mathbb{C}\langle Y^{1/\widehat{d}}\rangle[X]$, for $i = 1, \ldots, r$, satisfying

$$F(X,Y) = G_1^{(k)}(X,Y) \cdots G_r^{(k)}(X,Y) \mod S_{k+1}, \tag{7.7}$$

and $G_i^{(k)}(X,Y) \equiv G_i^{(0)}(X,Y) \mod S_1$, for all $i = 1, \ldots, r$.

```
> PS := PowerSeries([X, Y]) :
  with(PS) :
  UPoPS := UnivariatePolynomialOverPowerSeries([X, Y], Z) :
  with(UPoPS) :
  u := UPoPS:-FromListOfPolynomials([Y, 1, X + 1]);
  UPoPS:-PolynomialPart(u, 2);
  (p, alpha) := UPoPS:-WeierstrassPreparation(u, 2);
  UPoPS:-PolynomialPart(p, 2);
  UPoPS:-PolynomialPart(alpha, 2);
```

$$u := polynomial_over_power_series$$
$$Y + Z + (X + 1)\, Z^2$$
$$p, \alpha := polynomial_over_power_series,\ polynomial_over_power_ser$$
$$Y^2 + Y + Z$$
$$-X\,Y - Y^2 - Y + 1 + (X + 1)\, Z$$

```
> P := PowerSeries([y, z]) :
  U := UnivariatePolynomialOverPowerSeries([y, z], x) :
  poly := y·x^3 + (-2 · y + z + 1)·x + y :
  U:-ExtendedHenselConstruction(poly, [0, 0], 3);
```

$$\left[\left[x = \frac{-RootOf(_Z^2 + y) + RootOf(_Z^2 + y)\, y - \frac{1}{2}\, RootOf(_Z^2 + y)\, z + \frac{1}{2}\, y^2}{y}\right], \left[x = \frac{RootOf(_Z^2 + y) - RootOf(_Z^2 + y)\, y + \frac{1}{2}\, RootOf(_Z^2 + y)\, z + \frac{1}{2}\, y^2}{y}\right], [x = -y]\right]$$

Figure 7.2: On the right: Weierstrass Preparation Factorization for a univariate polynomial with multivariate power series coefficients. On the Left: Extended Hensel construction applied to a trivariate polynomial for computing its absolute factorization.

```
> P := PowerSeries([y]) :
  U := UnivariatePolynomialOverPowerSeries([y], x) :
  poly := y·x^3 + (-2 · y + 1)·x + y :
  OutputFlag :: name :='parametric':
  parametricVar :: name := T :
  iter := 3 :
  verificationFlag :: boolean := true :
  U:-ExtendedHenselConstruction(poly, 0, iter, OutputFlag, parametricVar, verificationFlag);
```

$$\left[\left[y = T^2,\ x = \frac{RootOf(_Z^2 + 1)\, T - T^3\, RootOf(_Z^2 + 1) + \frac{1}{2}\, T^4}{T}\right], \left[y = T^2,\ x = \frac{-RootOf(_Z^2 + 1)\, T + T^3\, RootOf(_Z^2 + 1) + \frac{1}{2}\, T^4}{T}\right], \left[y = T^2,\ x = -T^3\right]\right]$$

Figure 7.3: Extended Hensel construction applied to a bivariate polynomial for computing its Puiseux parametrizations around the origin.

7.2.3 The `PowerSeries` Library

The `PowerSeries` library consists of two modules, dedicated respectively to multivariate power series over the algebraic closure of $\mathbb{Q}$, and univariate polynomials with multivariate power series coefficients. Figure 7.2 illustrates *Weiertrass Preparation Factorization*. The command `PolynomialPart` displays all the terms of a power series (or a univariate polynomial over power series) up to a specified degree. In fact, each power series is represented by its terms that have been computed so far together with a program for computing the next ones. A command like `WeiertrassPreparation` computes the terms of the factors p and α up to the specified degree; moreover, the encoding of p and α contains a program for computing their terms in higher degree. Figures 7.2 and 7.3 illustrate the *Extended Hensel Construction* (EHC)[2] For the case of an input bivariate polynomial, see Figure 7.3, this coincides with the Newton-Puiseux algorithm, thus computing the Puiseux parametrizations of a plane curve about a point; this functionality is at the core of the `LimitPoints` command. For the case of a univariate polynomial with multivariate polynomial coefficients, the EHC is a weak version of Jung-Abhyankar Theorem.

[2]The factorization based on Hensel Lemma is in fact a weaker construction since: (1) the input polynomial must be monic and (2) the output factors may not be linear.

7.3 Applications

In this section we are concerned with two smooth maps $\Phi, \Psi : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2$ whose state variables and bifurcation parameter are denoted by (x, y) and λ, respectively. Since the Jacobian matrix of each map is not full rank at the origin, the Implicit Function Theorem fails at solving (x, y) as a function of λ locally around the origin. This causes bifurcations to reside in local zeros of each singular smooth map. We recall that these bifurcations are treated via first applying the Lyapunov-Schmidt reduction to a singular smooth map ending up with a reduced map of the form (7.2) and then passing the result through singularity theory techniques. Here, we follow the same approach except that we employ the `ExtendedHenselConstruction` command to compute the reduced map. The latter factorizes one of the equations around the origin and the resulting real branches that go through the origin are plugged in the other one to obtain the desired map (7.2). Once the map is computed we use the concept of recognition problem to identify the type of singularity.

7.3.1 The Pitchfork Bifurcation

In spite of simple structure, the pitchfork bifurcation is highly observed in physical phenomena mostly in the presence of symmetry breaking. For instance, [17] reports on *spontaneous mirror-symmetry breaking through a pitchfork bifurcation in a photonic molecule made up of two coupled photonic-crystal nanolasers*. Furthermore, authors in [26] study the pitchfork bifurcation arising in LugiatoLefever (LL) equation which is a model for a *passive Kerr resonator in an optical fiber ring cavity*. Finally, [13, Example 4.1] captures pitchfork bifurcation while analyzing the local bifurcations of Chua's circuit. Here, we consider the exercise 3.2 on [16, Page 34]. Suppose that $\Phi : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2$ is defined by $\begin{pmatrix} \Phi_1 \\ \Phi_2 \end{pmatrix}$ where

$$\begin{aligned} \Phi_1(x, y, \lambda) &= 2x - 2y + 2x^2 + 2y^2 - \lambda x \\ \Phi_2(x, y, \lambda) &= x - y + xy + y^2 - 3\lambda x. \end{aligned} \tag{7.8}$$

To obtain the reduced system g in (7.2) we pass Φ_1 to the `ExtendedHenselConstruction` giving rise to the branches in Figure 7.4. Note that the second branch is not of interest as it does not pass the origin. Substituting the first branch into Φ_2, modulo monomials of degree ≥ 4, results in

$$g(y, \lambda) = 2y^3 - \frac{5}{2}y\lambda + \frac{9}{2}y^2\lambda - \frac{5}{4}y\lambda^2. \tag{7.9}$$

Given g in (7.9), the low order terms and the intrinsic generators are determined by $\mathbb{R}\{1, y, \lambda, y^2\}$ and $\{y\lambda, y^3\}$, respectively. Thus, Theorem 7.2.1 implies that g satisfies the recognition problem for pitchfork

$$f(0,0) = \frac{\partial}{\partial y} f(0,0) = \frac{\partial}{\partial \lambda} f(0,0) = \frac{\partial^2}{\partial y^2} f(0,0) = 0$$

$$\frac{\partial}{\partial y}\frac{\partial}{\partial \lambda} f(0,0) \neq 0, \quad \frac{\partial^3}{\partial y^3} f(0,0) \neq 0$$

This proves that the original system Φ has pitchfork singularity located at the origin.

```
> P := PowerSeries([y, lambda]) :
> U := UnivariatePolynomialOverPowerSeries([y, lambda], x) :
> poly := 2·x − 2·y + 2·x^2 + 2·y^2 − lambda·x :
> U:-ExtendedHenselConstruction(poly, [0, 0], 4);
```

$$\left[\left[y=0, \lambda=0, x=-2y^2+\frac{1}{2}y\lambda+y+4y^3-2y^2\lambda+\frac{1}{4}y\lambda^2+\frac{1}{8}y\lambda^3+8y^3\lambda-\frac{7}{4}y^2\lambda^2-12y^4\right], \left[y=0, \lambda=0, x=2y^2-\frac{1}{2}y\lambda-y+\frac{1}{2}\lambda-1-4y^3+2y^2\lambda-\frac{1}{4}y\lambda^2-\frac{1}{8}y\lambda^3-8y^3\lambda+\frac{7}{4}y^2\lambda^2+12y^4\right]\right]$$

Figure 7.4: EHC applied to $\Phi_1(x, y, \lambda)$.

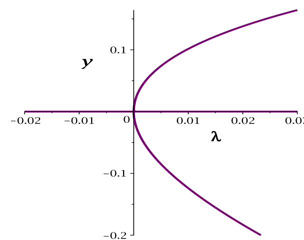

Figure 7.5: Pitchfork bifurcation diagram associated with g in Equation (7.9).

7.3.2 The Winged Cusp Bifurcation

The *winged cusp bifurcation problem* is defined by the equation $x^3 + \lambda^2 = 0$ and its corresponding bifurcation diagram $\{(x, \lambda) \mid x^3 + \lambda^2 = 0\}$ is exhibited via Figure 7.6. Singularity theory tools

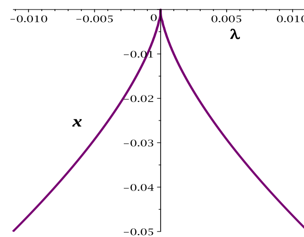

Figure 7.6: The winged cusp bifurcation diagram.

have been utilized in the area of chemical engineering with the aim of studying the solutions of the continuous flow stirred tank reactor (CSTR) model. This study proves that the winged cusp bifurcation is the normal form for describing the *organizing center* of the bifurcation diagrams of the model produced by numerical methods. It, further, unravels more bifurcation diagrams that have not been reported through these numerical methods; see [15,16,31,35]. Now assume

that $\Psi : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2$ is given by $\begin{pmatrix} \Psi_1 \\ \Psi_2 \end{pmatrix}$ where

$$\begin{aligned} \Psi_1(x, y, \lambda) &= -2x + 3y + \lambda^2 + y^3 + x^4 \\ \Psi_2(x, y, \lambda) &= 2x - 3y + y^2\lambda + x^3. \end{aligned} \tag{7.10}$$

Applying the `ExtendedHenselConstruction` to Ψ_2 leads to the branches in Figure 7.7.

```
> P := PowerSeries([y, lambda]) :
> U := UnivariatePolynomialOverPowerSeries([y, lambda], x) :
> poly := 2·x - 3·y + y^2·lambda + x^3 :
> U:-ExtendedHenselConstruction(poly, [0, 0], 3);
[[y = 0, λ = 0, x = 3/2 y - 1/2 y^2 λ - 27/16 y^3], [y = 0, λ = 0, x = -RootOf(_Z^2 + 2) - 3/4 y
   - 27/64 y^2 RootOf(_Z^2 + 2) + 1/4 y^2 λ + 27/32 y^3], [y = 0, λ = 0, x = RootOf(_Z^2 + 2) - 3/4 y
   + 27/64 y^2 RootOf(_Z^2 + 2) + 1/4 y^2 λ + 27/32 y^3]]
```

Figure 7.7: EHC applied to $\Psi_2(x, y, \lambda)$.

Substituting the first branch into Ψ_1, modulo monomials of degree ≥ 4, yields

$$g(y, \lambda) = \frac{35}{8} y^3 + \lambda^2 + y^2\lambda. \tag{7.11}$$

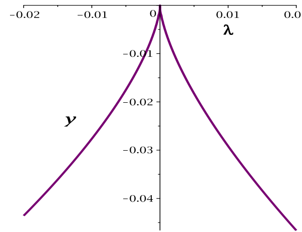

Figure 7.8: Bifurcation diagram associated with g in (7.11).

As $\{1, y, \lambda, y^2, y\lambda\}$ spans the space of low order terms and intrinsic generators are $\{\lambda^2, y^3\}$, Theorem 7.2.1 guarantees that g satisfies the recognition problem for the winged cusp

$$f(0,0) = \frac{\partial}{\partial y} f(0,0) = \frac{\partial}{\partial \lambda} f(0,0) = \frac{\partial^2}{\partial y^2} f(0,0) = \frac{\partial}{\partial y}\frac{\partial}{\partial \lambda} f(0,0) = 0$$

$$\frac{\partial^2}{\partial \lambda^2} f(0,0) \neq 0, \quad \frac{\partial^3}{\partial y^3} f(0,0) \neq 0$$

Chapter 8

Conclusions and Future Work

Throughout this thesis 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 [32] 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 opportunities 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.

Bibliography

[1] Brandt Alexander, Mahsa Kazemi, and Marc Moreno Maza. Power series arithmetic with the BPAS library. *To Appear in Lecture Notes in Computer Scienc*, 2020.

[2] 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.

[3] Parisa Alvandi, Masoud Ataei, and Marc Moreno Maza. On the extended hensel construction and its application to the computation of limit points. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC 17, pages 13–20, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Parisa Alvandi, Mahsa Kazemi, and Marc Moreno Maza. Computing limits of real multivariate rational functions. In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, pages 39–46, New York, NY, USA, 2016. ACM.

[5] 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.

[6] 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`.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] Majid Gazor and Mahsa Kazemi. A userguide for singularity, March 2018.

[13] Majid Gazor and Mahsa Kazemi. Normal form analysis of z2-equivariant singularities. *International Journal of Bifurcation and Chaos*, 29(2):1950015–1–1950015–20, 2019.

[14] Majid Gazor and Mahsa Kazemi. Singularity: A maple Library for Local Zero Bifurcation Control of Scalar Smooth Maps. *Journal of Computational and Nonlinear Dynamics*, 15(1), 2019.

[15] M. Golubitsky and B. L. Keyfitz. A qualitative study of the steady-state solutions for a continuous flow stirred tank chemical reactor. *SIAM J. Math. Anal.*, 11(2):316–339, 1980.

[16] Martin Golubitsky and David G. Schaeffer. *Singularities and groups in bifurcation theory. Vol. I*, volume 51 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 1985.

[17] Philippe Hamel, Samir Haddadi, Fabrice Raineri, Paul Monnier, Gregoire Beaudoin, Isabelle Sagnes, Ariel Levenson, and Alejandro M. Yacomotti. Spontaneous mirror-symmetry breaking in coupled photonic-crystal nanolasers. *Nature Photonics*, 9:311–315, 2015.

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

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

[20] 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.

[21] Isabel Labouriau. *Applications of singularity theory to neurobiology*. PhD Thesis, Warwick University, 1984.

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

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

[24] Michael B. Monagan and Paul Vrbik. Lazy and forgetful polynomial arithmetic and applications. In *Computer Algebra in Scientific Computing, 11th International Workshop, CASC 2009, Proceedings*, pages 226–239, 2009.

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

[26] J Rossi, R Carretero-González, P G Kevrekidis, and M Haragus. On the spontaneous time-reversal symmetry breaking in synchronously-pumped passive kerr resonators. *Journal of Physics A: Mathematical and Theoretical*, 49(45):455201, 2016.

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

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

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

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

[31] A. Uppal, W. H. Ray, and A. B. Poore. The classification of the dynamic behavior of continuous stirred tank reactorsinfluence of reactor residence time. *J. Chemical Engineering Science*, 31(3):205–214, 1976.

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

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

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

[35] Y. V. Zeldovich and U. A. Zisin. On the theory of thermal stress. flow in an exothermic stirred reactor, ii.study of heat loss in a flow reactor. *Journal of Technical Physics*, 11(6):501–508, 1941.

Curriculum Vitae

Name:	Mahsa Kazeminooreddinvand
Education	
	University of Western Ontario London, ON Sep 2018 – Jul 2020 M.Sc. in Computer Science
	Isfahan University of Technology Isfahan, Iran Sep 2012 – Jun 2018 Ph.D. in Applied Mathematics (Dynamical Systems)
	Shiraz University Shiraz, Iran Sep 2008 – Sep 2010 MSc in Applied Mathematics (Dynamical Systems)
Related Work & Experience:	
	Maplesoft, Waterloo June 2020 - present R & D Intern
	The University of Western Ontario Sep 2018 - Jul 2020 Teaching Assistant
	Maplesoft, Waterloo May 2019 - Sep 2019
	The University of Western Ontario Nov 2015 - Nov 2016 Visiting Scholar
	Isfahan University of Technology Sep 2012 - Jun 2018 Teaching Assistant and Course Instructor
	Qatar University Jun 2010 Visiting Scholar

Publications:

- Alexander Brandt, Mahsa Kazemi and Marc Moreno Maza, *Power Series Arithmetic with the* `BPAS` *Library*, To Appear in Lecture Notes in Computer Science, 2020.
- Majid Gazor and Mahsa Kazemi, `Singularity` *: A Maple library for local zero bifurcation control of scalar smooth maps* J. Computational and Nonlinear Dynamics, Transactions of the American Society of Mechanical Engineers (ASME) 15(1) (2019): 011001–0110010.
- Mahsa Kazemi and Marc Moreno Maza, *Detecting Singularities Using the* `PowerSeries` *Library* Communications in Computer and Information Science, (2020): 145–155.
- Amir Hashemi and Mahsa Kazemi, *Parametric standard bases and its applications* Lecture Notes in Computer Science 1166 (2019): 179–196.
- Majid Gazor and Mahsa Kazemi, *Normal form analysis of Z_2-equivariant singularities* International Journal of Bifurcation and Chaos, 29 (2019): 1950015–1950035.
- 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* Journal of Symbolic Computation, 98 (2019): 120–162.
- Majid Gazor, Amir Hashemi and Mahsa Kazemi, *Groebner bases and multi-dimensional persistent bifurcation diagram classifications* ACM Communications in Computer Algebra, 52(4) (2018): 120–122.
- Majid Gazor and Mahsa Kazemi, *A userguide for Singularity* arXiv preprint arXiv:1601.00268, (2016).
- Parisa Alvandi, Mahsa Kazemi and Marc Moreno Maza, *Computing limits of real multivariate rational functions* Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation (ISSAC), (2016): 39–46.
- Parisa Alvandi, Mahsa Kazemi and Marc Moreno Maza, *Computing limits with the RegularChains and PowerSeries libraries: from rational functions to zariski closure* ACM Communications in Computer Algebra, 50(3) (2016): 93–96.
- Majid Gazor and Mahsa Kazemi, *Z_2 -equivariant standard bases for submodules associated with Z_2-equivariant singularities* ACM Communications in Computer Algebra, 50(4) (2016): 170–172.