A new bivariate Hensel lifting algorithm for n factors

by

Garrett Leonard Ross Paluck

B.Sc., Thompson Rivers University, 2015

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
Department of Mathematics
Faculty of Science

SIMON FRASER UNIVERSITY
Summer 2019

Approval

Name:	**Garrett Leonard Ross Paluck**
Degree:	**Master of Science (Pure Mathematics)**
Title:	**A new bivariate Hensel lifting algorithm for n factors**
Examining Committee:	**Chair:** Tom Archibald Professor **Michael Monagan** Senior Supervisor Professor **Jonathan Jedwab** Supervisor Professor **Stephen Choi** Internal Examiner Professor
Date Defended:	**August 13, 2019**

Abstract

We present a new algorithm for performing Linear Hensel Lifting of bivariate polynomials over the finite field $\mathbb{F}_p$ for some prime p. Our algorithm lifts n monic, univariate polynomials to recover the factors of a polynomial $A(x, y) \in \mathbb{F}_p[x, y]$ which is monic in x, and bounded by degrees $d_x = \deg(A, x)$ and $d_y = \deg(A, y)$. Our algorithm improves upon Bernardin's algorithm in [2] and reduces the number of arithmetic operations in $\mathbb{F}_p$ from $O(n\, d_x^2\, d_y^2)$ to $O(d_x^2\, d_y + d_x\, d_y^2)$ for $p \geq d_x$. Experimental results in C verify that our algorithm compares favorably with Bernardin's for large degree polynomials.

Keywords: Hensel Lifting, polynomial factorization, modular methods, arithmetic operations, bivariate polynomials

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

We have created a new Linear Hensel Lifting algorithm that calculates the factorization of a polynomial $A(x, y)$ over a finite field $\mathbb{F}_p$ for some prime p by lifting $n \geq 2$ bivariate factors from univariate images. Our algorithm is a variation of the Linear Hensel Lifting algorithm for factoring bivariate polynomials. Let A have degree d_x and d_y in variables x and y respectively. Our algorithm uses $O(d_x^2\, d_y + d_x\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$ to do the Hensel Lifting provided $p \geq d_x$.

We begin with a precise statement of what we want to compute. Suppose we are given some prime p and a polynomial $A \in \mathbb{F}_p[x, y]$ that is monic in x. A must also be *square-free*, that is, $\nexists\, b$ with $\deg(b, x) > 0$ such that $b^2 | A$. Suppose we choose some $\alpha \in \mathbb{F}_p$ and obtain the factorization $A(x, \alpha) = \prod_{i=1}^{n} f_{i,0}$ in which $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ are monic and pairwise relatively prime. One way to factor $A(x, \alpha)$ over $\mathbb{F}_p$ is to use the Cantor-Zassenhaus algorithm [3]. We note that it is possible for A to be square-free but for $A(x, \alpha)$ to not be square-free. We discuss this in Section 1.2.

Bivariate Hensel Lifting aims to construct monic, bivariate polynomials $f_1^{(k)}, f_2^{(k)}, ..., f_n^{(k)} \in \mathbb{F}_p[x, y]$ where

$$f_i^{(k)} = f_i^{(k-1)} + f_{i,k-1}(y - \alpha)^{(k-1)} \text{ and } f_i^{(1)} = f_{i,0}$$

such that

$$\begin{array}{ll} (1) & \forall i : f_i^{(k)} \equiv f_{i,0} \pmod{y - \alpha}, \text{ and} \\ (2) & A \equiv \prod_i f_i^{(k)} \pmod{(y - \alpha)^k}. \end{array}$$

If k is sufficiently large, the $f_i^{(k)}$ obtained can be used to compute a factorization of A over $\mathbb{F}_p$. We will stop Hensel Lifting when $\prod_i f_i^{(k)} = A(x, y)$ or when $\sum_{i=1}^{n} \deg(f_i^{(k)}, y) > \deg(A, y) = d_y$. If the latter conditions occur, then we clearly have an incorrect factorization of A.

Example 1.1. *Our algorithm will find a factorization of the bivariate polynomial*

$$A(x,y) = x^3 + 6y^4 + (-6x+8)y^3 + (4x^2 - 8x + 2)y^2 + (-x^2 + 6x + 4)y - 5x^2 - 6x$$

over the field $\mathbb{F}_{17}$*. If we choose the element* $\alpha = 3$ *and obtain the initial monic factors* $f_{1,0} = x+7, f_{2,0} = x+6$*, and* $f_{3,0} = x-2$ *such that* $A - f_{1,0}f_{2,0}f_{3,0} \equiv 0 \pmod{y-3}$*, then our algorithm lifts* $f_{1,0}, f_{2,0}$*, and* $f_{3,0}$ *to factorize* A *as*

$$A(x,y) = \underbrace{(x + 2(y-3) + 7)}_{f_1} \times \underbrace{(x + 4(y-3)^2 - (y-3) + 6)}_{f_2} \times \underbrace{(x + 5(y-3) - 2)}_{f_3}$$

△

We restrict ourselves to the case of bivariate polynomials over a finite field. Bivariate polynomials are common in practice. Moreover, as we outline below, state of the art algorithms for factoring polynomials in more than two variables rely on multiple factorizations of bivariate polynomials [9], [10], [18]. For these reasons, it is important to have a fast way of lifting bivariate polynomials.

One way to factor a multivariate polynomial in $\mathbb{Z}[x_1, ..., x_m]$ is to factor a univariate image in $\mathbb{Z}[x]$, then Hensel lift the images to factors in $\mathbb{Z}[x_1, ..., x_m]$. This method was developed by Rothschild and Wang in [23]. It is used in the Maple and Magma computer algebra systems. The computer algebra system Singular uses a different approach. In [12], Lee and Pfister factor a bivariate image, then lift the resulting bivariate images to the factors in $\mathbb{Z}[x_1, ..., x_m]$. To compute the bivariate images, they use Hensel lifts in $\mathbb{F}_p[x, y]$.

In [18], Monagan and Tuncer developed an algorithm to use Hensel Lifting to find a factorization of multivariate polynomials in $\mathbb{Z}[x_1, ..., x_m]$. They use evaluation to reduce the multivariate polynomial to many bivariate images, then use bivariate Hensel Lifting to factor the images, and finally use interpolation to recover the factorization of the original multivariate polynomial. Their method requires an algorithm which can perform bivariate Hensel Lifting quickly since they may have to do thousands of bivariate Hensel lifts if the factors of A have many terms or many variables of high degree. Their algorithm makes repeated use of bivariate Hensel Lifting which provides clear motivation for finding efficient algorithms for performing bivariate Hensel lifts.

We use a dense lifting approach which is most effective for bivariate polynomials. Factorization algorithms for sparse multivariate polynomials (polynomials where most of the coefficients of its monomials are zero) have been presented in [22], [9], [17], [21], [23]. Only as the number of variables grows does it becomes increasingly important to use sparse techniques to prevent exponential behaviour in the number of variables.

Our algorithm is based on an algorithm developed by Laurent Bernardin in 1998 which we will refer to as Bernardin's algorithm. Bernardin's algorithm was originally published in the proceedings of ISSAC 1998[2] and presented a parallel algorithm for performing Linear Hensel Lifting of bivariate polynomials with n factors over a finite field $\mathbb{F}_p$. The sequential version of Bernardin's algorithm uses $O(n\,d_x^2\,d_y^2)$ arithmetic operations in $\mathbb{F}_p$ for lifting the initial univariate images to the bivariate factors. Bernardin's algorithm is currently implemented in Maple's "modp2/Factors" library and is called to factor certain bivariate polynomials over finite fields.

We make two major improvements to Bernardin's algorithm. We improve the bottleneck and we apply stricter degree bounds to some subroutines of Bernardin's algorithm. Both of these improvements result in a reduction of the asymptotic complexity of Bernardin's algorithm. We will describe the bottleneck when we cover Bernardin's algorithm in detail in Section 1.3. In brief, the bottleneck of Bernardin's algorithm is its method for calculating a certain polynomial, $\Delta \in \mathbb{F}_p[x]$, which requires calculating the product of many univariate polynomials and adding them together. As we will show in Chapter 3, we use polynomial evaluation, interpolation, and point-wise multiplication in $\mathbb{F}_p$ to calculate Δ more efficiently. This improvement reduces the number of arithmetic operations done by Bernardin's algorithm from $O(n\,d_x^2\,d_y^2)$ to $O(n\,d_x^2\,d_y + n\,d_x\,d_y^2)$. Through further analysis, we have found that by applying strict degree bounds to all intermediate calculations and sub-algorithms in our algorithm, we can reduce the complexity by an additional factor of n to $O(d_x^2\,d_y + d_x\,d_y^2)$. Most notably, we use the fact that $d_x = \sum_i \deg(f_i^{(k)}, x)$ and $d_y \leq \sum_i \deg(f_i^{(k)}, y)$ to reduce the complexity. The $n = 2$ case is considered by Monagan in [16]. This thesis considers the case when $n \geq 2$.

To properly present our algorithm, which discovers a factorization of A, and its complexity, we will break this thesis into several chapters. Chapter 1 will begin by chronicling the historical versions of Hensel Lifting in $\mathbb{Z}[x]$ to factor univariate polynomials into two factors. We then expand on the classical algorithm to show how Hensel Lifting can be used to factor a bivariate polynomial in $\mathbb{F}_p[x, y]$ into n bivariate factors. We conclude Chapter 1 with an in-depth analysis of Bernardin's algorithm for factoring bivariate polynomials in $\mathbb{F}_p[x, y]$. Chapter 2 will cover well-known algorithms that we shall require. Chapter 3 will outline how and why our algorithm works, and calculate the number of arithmetic operations in $\mathbb{F}_p$ that it uses. Finally, Chapter 4 will present some benchmarks for the implementation of our algorithm and Bernardin's algorithm in the C programming language. We will verify, through experimentation, that our algorithm both outperforms Bernardin's and has a cubic complexity. In summary, this thesis will show why our algorithm is superior to Bernardin's algorithm in both theory and practice.

Our main technical result is the following theorem.

Theorem 1.2. *Let $A \in \mathbb{F}_p[x,y], d_x = \deg(A,x) > 1, d_y = \deg(A,y) > 1$. Suppose $A = f_1 f_2 \ldots f_n$ and we are given pairwise relatively prime images $f_i(x,\alpha)$ of the factors f_i for some $\alpha \in \mathbb{F}_p$. If $p \geq d_x$, we can compute $f_1 f_2 \ldots f_n$ in $O(d_x^2\, d_y + d_x\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$ using space for $O(n\, d_x\, d_y)$ elements of $\mathbb{F}_p$.*

We cover benchmarks in detail in Chapter 4. Table 1.1 shows timings for using both our algorithm and Bernardin's algorithm for bivariate Hensel Lifting $n = 4$ factors in $\mathbb{F}_p[x,y]$ with $p = 2^{31}-1$ and $d = d_x = d_y$. The factors f_1, f_2, f_3 and f_4 have the form $x^{d/4} + \sum_{i=0}^{\frac{d}{4}} (\sum_{j=0}^{\frac{d}{4}-1} c_{ij} x^j) y^i$ where the coefficients c_{ij} are chosen at random from $[0,p)$. We then input $\alpha = 3, A = f_1 \times f_2 \times f_3 \times f_4$, $f_{1,0} = f_1(x,\alpha), f_{2,0} = f_2(x,\alpha), f_{3,0} = f_3(x,\alpha)$, and $f_{4,0} = f_4(x,\alpha)$ to the Hensel Lifting algorithms. The speedup is a comparison between the time it took to compute our algorithm compared to Bernardin's.

	Our Cubic Algorithm	Bernardin's Algorithm	
d	Time(ms)	Time(ms)	Speedup
16	0.13	0.11	0.85
32	0.36	0.43	1.19
64	1.49	3.38	2.27
128	7.77	34.40	4.43
256	45.38	362.85	8.00
512	324.19	4,319.31	13.32
1024	2,502.76	55,716.30	22.26
2048	18,017.49	782,982.80	43.46
4096	128,211.01	11,647,207.28	90.84
8192	963,335.81	NA	-

Table 1.1: Hensel Lifting timings for $\mathbb{F}_p[x,y]$ with $p = 2^{31}-1$ and $n = 4$ factors. NA = not attempted

1.1 Historical Hensel Lifting in $\mathbb{Z}[x]$

We begin our discussion with the history of the Hensel Lifting algorithm. The idea is based on Hensel's Lemma which was penned by Kurt Hensel (1861-1941). Zassenhaus [25] was the first to apply Hensel's Lemma to factor polynomials in $\mathbb{Z}[x]$. The algorithm was then adapted by Miola and Yun [15] to create an algorithm for performing Linear Hensel Lifting and applied to compute polynomial gcds in $\mathbb{Z}[x]$. Knuth in [11] suggested that if p is sufficiently large, then Hensel's Lemma could be used to find the gcd of two polynomials but did not attribute this observation to Zassenhaus. A description of Hensel Lifting may be found in *Modern Computer Algebra*[6] and *Algorithms for Computer Algebra*[7]. All modern Computer Algebra systems, such as Maple or Magma, factor polynomials in $\mathbb{Z}[x]$ use Hensel Lifting.

In [2], Bernardin modified the Linear Hensel Lifting algorithm of Miola and Yun [15] to factor polynomials in $\mathbb{F}_p[x, y]$. Miola and Yun covered only the two factor case as that is all that is needed for computing gcds. Bernardin covered the $n \geq 2$ factor case.

1.1.1 Solving a polynomial Diophantine equation

Before we can discuss Hensel's Lemma or Hensel Lifting, we need to be able to solve a particular polynomial Diophantine equation for $n = 2$ factors. Given a prime p, and $a(x), b(x), c(x) \in \mathbb{Z}_p[x]$, we must solve

$$\sigma a + \tau b = c \tag{1.1}$$

for $\sigma(x), \tau(x) \in \mathbb{Z}_p[x]$ where we are given $\gcd(a, b) = 1$ and require $\deg(\sigma, x) < \deg(b, x)$. The degree constraint on σ imposes uniqueness. Solving (1.1) is necessary for lifting the factors in each iteration of Hensel's Lemma. We present the necessary theorem as Theorem 1.3. This theorem corresponds to Theorem 2.6 in [7]. To demonstrate that this theorem holds for any field, we present it for a general field F .

Theorem 1.3. *Let $F[x]$ be the Euclidean domain of univariate polynomials over a field F. Let $a(x), b(x) \in F[x]$ be given nonzero polynomials and let $g(x) = \gcd(a, b) \in F[x]$. Then for any given polynomial $c(x) \in F[x]$ such that $g|c$ there exist unique polynomials $\sigma(x), \tau(x) \in F[x]$ such that*

$$\sigma a + \tau b = c \text{ and} \tag{1.2}$$

$$\deg(\sigma, x) < \deg(b, x) - \deg(g, x). \tag{1.3}$$

Moreover, if $\deg(c, x) < \deg(a, x) + \deg(b, x) - \deg(g, x)$ then τ satisfies

$$\deg(\tau, x) < \deg(a, x) - \deg(g, x). \tag{1.4}$$

We discuss the polynomial Diophantine equation now because it is used in both Hensel's Lemma and the Hensel Lifting algorithm. We will discuss this theorem in more detail, including its proof of existence and uniqueness, as well as its complexity, later in Section 2.2.1. We will also prove that finding a solution to (1.1) requires $O(d^2)$ arithmetic operations in F where $\deg(a, x) \leq \deg(b, x) \leq d$ and $\deg(c, x) < \deg(a, x) + \deg(b, x)$. We will then cover the extension of the polynomial Diophantine equation for $n > 2$ factors directly following the $n = 2$ factor case in Section 2.2.2. We give an example of how to solve (1.1) for σ and τ as Example 1.4.

Example 1.4. *Let $F = \mathbb{Z}_{11}$ be the finite field with 11 elements. Let $a = 3x^2 + 4x + 6, b = 2x^2 + x + 3$, and $c = 6x + 1$. We will find $\sigma, \tau \in \mathbb{Z}_{11}[x]$ such that $\sigma a + \tau b = c$ and $\deg(\sigma, x) < \deg(b, x)$. Note that $\gcd(a, b) = 1$ in $\mathbb{Z}_{11}[x]$.*

We begin by applying the Extended Euclidean Algorithm (EEA) to solve

$$sa + tb = 1 \tag{1.5}$$

for $s, t \in \mathbb{Z}_{11}[x]$. The EEA finds $s = 2x + 2$ and $t = 8x$. We then multiply both sides of (1.5) by c to get

$$csa + ctb = (x^2 + 3x + 2)(3x^2 + 4x + 6) + (4x^2 + 8x)(2x^2 + x + 3) = 6x + 1.$$

As $\deg(cs, x) \not< \deg(b, x)$, we calculate $cs \div b$ using the classical division algorithm to obtain some quotient q and remainder r such that $cs = qb + r$. We get $q = 6$ and $r = 8x + 6$. We can now solve for σ and τ as follows:

$$c = csa + ctb = (qb + r)a + ctb = \underbrace{r}_{\sigma} a + \underbrace{(qa + ct)}_{\tau} b. \tag{1.6}$$

By applying (1.6), we obtain $\sigma = r = 8x + 6$ and $\tau = qa + ct = 10x + 3$. Note that now $\deg(\sigma, x) < \deg(b, x)$.

△

1.1.2 Hensel's Lemma

Hensel's Lemma, also known as Hensel's Lifting Lemma, is a famous result in algebraic number theory. It states that if a polynomial equation has a simple root modulo a prime number p, then this root corresponds to a unique root of the same equation modulo any higher power of p, which can be found by iteratively "lifting" the solution modulo successive powers of p. Since p-adic analysis is in some ways simpler than real analysis, there are relatively neat criteria guaranteeing a root of a polynomial. We present Hensel's Lemma as Theorem 1.5. The proof of Hensel's Lemma, given here, is a constructive proof which is commonly referred to as the *Hensel Construction.* Our proof of Hensel's Lemma corresponds to the proof of Theorem 6.2 in [7].

Theorem 1.5 (Hensel's Lemma)**.** *Let p be a prime in $\mathbb{Z}$ and let $A(x) \in \mathbb{Z}[x]$ be a given non-zero polynomial over the integers. Let $u^{(1)}, w^{(1)} \in \mathbb{Z}_p[x]$ such that*

$$A \equiv u^{(1)} w^{(1)} \pmod{p} \tag{1.7}$$

and $gcd(u^{(1)}, w^{(1)}) = 1$ in $\mathbb{Z}_p[x]$. Then for any integer $k > 1$, there exist polynomials $u^{(k)}, w^{(k)} \in \mathbb{Z}[x]$ such that

$$A \equiv u^{(k)} w^{(k)} \pmod{p^k} \tag{1.8}$$

and

$$u^{(k)} \equiv u^{(1)} \pmod{p}, \;\; w^{(k)} \equiv w^{(1)} \pmod{p}. \tag{1.9}$$

Proof. The proof is by induction on k. The case $k = 1$ is given. Assume for $k > 1$ that we have $u^{(k)}, w^{(k)} \in \mathbb{Z}[x]$ satisfying (1.8) and (1.9). Notice that (1.8) implies p^k divides $A - u^{(k)}w^{(k)}$. We define

$$c_k = \frac{A - u^{(k)}w^{(k)}}{p^k} \in \mathbb{Z}[x]. \tag{1.10}$$

Since, $u^{(1)}w^{(1)} \in \mathbb{Z}_p[x]$ are relatively prime, by Theorem 1.3 we can find unique polynomials $\sigma_k, \tau_k \in \mathbb{Z}_p[x]$ such that

$$\sigma_k w^{(1)} + \tau_k u^{(1)} = c_k \tag{1.11}$$

with

$$\deg(\sigma_k, x) < \deg(u^{(1)}, x). \tag{1.12}$$

Then by defining

$$u^{(k+1)} = u^{(k)} + \sigma_k p^k \text{ and } w^{(k+1)} = w^{(k)} + \tau_k p^k \tag{1.13}$$

we have by performing multiplication modulo p^{k+1}:

$$\begin{aligned}
u^{(k+1)}w^{(k+1)} &= (u^{(k)} + \sigma_k p^k)(w^{(k)} + \tau_k p^k) \\
&= u^{(k)}w^{(k)} + (\sigma_k w^{(k)} + \tau_k u^{(k)})p^k + \sigma_k \tau_k p^{2k} \\
&\equiv u^{(k)}w^{(k)} + (\sigma_k w^{(k)} + \tau_k u^{(k)})p^k \pmod{p^{k+1}}, \quad \text{by (1.9)} \\
&\equiv u^{(k)}w^{(k)} + c_k p^k \pmod{p^{k+1}}, \quad \text{by (1.11)} \\
&\equiv u^{(k)}w^{(k)} + (\frac{A - u^{(k)}w^{(k)}}{p^k})p^k + 0 \pmod{p^{k+1}}, \quad \text{by (1.10)} \\
&\equiv u^{(k)}w^{(k)} + A - u^{(k)}w^{(k)} \pmod{p^{k+1}} \\
&\equiv A \pmod{p^{k+1}}.
\end{aligned}$$

Thus (1.8) holds for $k + 1$. Also from (1.13), it is clear that

$$u^{(k+1)} \equiv u^{(k)} \pmod{p} \text{ and } w^{(k+1)} \equiv w^{(k)} \pmod{p}$$

and therefore since (1.9) holds for k, it also holds for $k + 1$.

□

Example 1.6. *Consider the problem of finding polynomials $u^{(3)}(x), w^{(3)}(x) \in \mathbb{Z}[x]$ with respect to $u^{(1)}(x), w^{(1)}(x) \in \mathbb{Z}_7[x]$ respectively according to Theorem 1.5.*

Let $p = 7, A(x) = x^4 + 24x^3 + 160x^2 + 229x - 84, u^{(1)}(x) = x^2 + 4x,$ and $w^{(1)}(x) = x^2 + 6x + 3$. Note that $A(x) \equiv u^{(1)}w^{(1)} \pmod{p}$. We will define

$$c_1 = \frac{A - u^{(1)}w^{(1)}}{p} = \frac{x^4 + 24x^3 + 160x^2 + 229x - 84 - (x^2+4x)(x^2+6x+3)}{7} = 2x^3 + 19x^2 + 31x - 12.$$

We solve the polynomial Diophantine equation

$$\sigma_1 w^{(1)} + \tau_1 u^{(1)} = c_1$$

for $\sigma_1, \tau_1 \in \mathbb{Z}_p[x]$ *such that* $\deg(\sigma_1, x) < \deg(u^{(1)}, x)$*. By applying the method described in Section 1.1.1, we find the unique solution* $\sigma_1 = x + 3$ *and* $\tau_1 = x + 6$*. We then update* $u^{(2)}$ *and* $w^{(2)}$ *as*

$$u^{(2)} = u^{(1)} + \sigma_1 p = (x^2 + 4x) + (x+3)7 = x^2 + 11x + 21 \quad \textit{and}$$

$$w^{(2)} = w^{(1)} + \tau_1 p = (x^2 + 6x + 3) + (x+6)7 = x^2 + 13x + 45.$$

We apply Theorem 1.5 again to find $u^{(3)}$ *and* $w^{(3)}$*. Let*

$$c_2 = \frac{A - u^{(2)}w^{(2)}}{p^2} = -x^2 - 11x - 21.$$

We next solve the polynomial Diophantine equation

$$\sigma_2 w^{(1)} + \tau_2 u^{(1)} = c_2$$

for $\sigma_2, \tau_2 \in \mathbb{Z}_p[x]$ *such that* $\deg(\sigma_2, x) < \deg(u^{(1)}, x)$*.*

By applying the method described in Section 1.1.1 again, we find the unique solution $\sigma_2 = 0$ *and* $\tau_2 = 6$*. We then update* $u^{(3)}$ *and* $w^{(3)}$ *as*

$$u^{(3)} = u^{(2)} + \sigma_2 p^2 = x^2 + 11x + 21 \quad \textit{and} \quad w^{(3)} = w^{(2)} + \tau_2 p^2 = x^2 + 13x + 290.$$

We have found $u^{(3)}$ *and* $w^{(3)}$ *such that* $u^{(3)} \equiv u^{(1)} \pmod{p}, w^{(3)} \equiv w^{(1)} \pmod{p}$*, and* $A \equiv u^{(3)}w^{(3)} \pmod{p^3}$*. Note: in our example the leading coefficients of* $u^{(k)}$ *and* $w^{(k)}$ *are 1 and remain 1 throughout. This is a consequence of the degree constants* $\deg(\sigma, x) < \deg(u^{(1)}, x)$ *and* $\deg(\tau, x) < \deg(w^{(1)}, x)$ *from Theorem 1.3.*

△

Corollary 1.7. *(Uniqueness of the Hensel Construction). In Theorem 1.5, if the given polynomial* $A(x) \in \mathbb{Z}[x]$ *is monic and if the relatively prime factors* $u^{(1)}, w^{(1)} \in \mathbb{Z}_p[x]$ *are monic, then for any integer* $k \geq 1$ *conditions (1.3), (1.4) uniquely determine the monic polynomial factors* $u^{(k)}, w^{(k)} \in$

$\mathbb{Z}_{p^k}[x]$. *In addition, the leading monic term of* $u^{(k)}$ *and* $w^{(k)}$ *match the leading terms of* $u^{(1)}$ *and* $w^{(1)}$ *respectively.*

A proof of Corollary 1.7 can be found in [7]. All variations of Hensel Lifting come from Hensel's Lemma. We will next show how to adapt Hensel's Lemma to define Hensel Lifting for 2 factors in $\mathbb{Z}[x]$.

1.1.3 Hensel Lifting in $\mathbb{Z}[x]$ for two factors

In this section, we discuss the method needed to perform Hensel Lifting for two factors in $\mathbb{Z}[x]$. We begin by describing the p-adic representation of polynomials as we will be lifting univariate polynomials in terms of a prime p. We then explain the method of Hensel Lifting. This will be followed by the statement of the classical Hensel Lifting algorithm in $\mathbb{Z}[x]$ for two factors. Finally, we will finish this section by calculating the complexity of the Hensel Lifting algorithm.

Consider the problem of inverting the modular homomorphism $\phi_p : \mathbb{Z}[x] \to \mathbb{Z}_p[x]$. The starting point in the development of Hensel Lifting is to consider yet another representation for integers and polynomials. The approach is based on constructing an integer solution v in its p-adic (or base p) representation. Let

$$v = v_0 + v_1 p + \cdots + v_{n-1} p^{n-1} \tag{1.14}$$

where p is a positive prime integer, $n \in \mathbb{N}$ is large enough that $p^n > 2|v|$, and $v_i \in \mathbb{Z}_p$ for $0 \le i < n$. The p-adic representation can be developed using either the positive or the symmetric representation of $\mathbb{Z}_p$. If the positive representation is used then (1.14) is simply the familiar *radix p representation* of the nonnegative integer v (and it is sufficient for n to be such that $p^n > v$). However, the symmetric representation is useful in practice because it allows the integer v to be negative.

Lemma 1.8. *Let* p *be a prime and* u *be an integer such that* $-\frac{p^n}{2} < v < \frac{p^n}{2}$. *There exist unique integers* v_i $(0 \le i \le n-1)$ *such that* $v = \sum_{i=0}^{n-1} v_i\, p^i$ *and* $-\frac{p}{2} < v_i < \frac{p}{2}$.

Proof. We will not provide a proof of existence for this lemma. We will provide a method for finding the p-adic representation for an integer v directly following this proof.

(Uniqueness) Suppose $v = v_0 + v_1 p + v_2 p^2 + ... + v_{n-1} p^{n-1}$ and $v = v'_0 + v'_1 p + v'_2 p^2 + ... + v'_{n-1} p^{n-1}$ satisfying $-\frac{p}{2} < v_i < \frac{p}{2}$ and $-\frac{p}{2} < v'_i < \frac{p}{2}$. Then

$$v - v = 0 = (v_0 - v'_0) + (v_1 - v'_1)p + ... + (v_{n-1} - v'_{n-1})p^{n-1}. \tag{1.15}$$

Reducing mod p, we have $0 \equiv (v_0 - v'_0) \pmod p$ which implies

$$p|(v_0 - v_0').$$

Now, the conditions on v_0 and v_0' imply $-p < v_0 - v_0' < p$ and therefore $v_0 - v_0' = 0$ and so $v_0 = v_0'$. Dividing (1.15) by p we obtain

$$0 = (v_1 - v_1') + (v_2 - v_2')p + \cdots + (v_{n-1} - v_{n-1}')p^{n-2}. \tag{1.16}$$

Repeating the above argument we show that $v_1 = v_1'$, then repeating again we show that $v_i = v_i'$ for all i, thus we have uniqueness for the symmetric p-adic representation.

□

There is a simple procedure for developing the p-adic representation for a given integer v. Firstly, we see from equation (1.14) that $v \equiv v_0 \pmod p$, so using the modular mapping $\phi_p(a) = \mathrm{rem}(a, p)$ we have

$$v_0 = \phi_p(v). \tag{1.17}$$

For the next p-adic coefficient v_1, note that $v - v_0$ must be divisible by p, and from equation (1.14) it follows that

$$\frac{v - v_0}{p} = v_1 + v_2 p + ...v_n p^{n-1}.$$

Notice $(v - v_0)/p$ is the quotient q_1 of $v \div p$. Next we obtain v_1 from

$$v_1 = \phi_p\left(\frac{v - v_0}{p}\right) = \phi_p(q_1).$$

Again, for the next p-adic coefficient v_2, note that $q_1 - v_1$ must be divisible by p, so it follows that

$$\frac{q_1 - v_1}{p} = v_2 + v_3 p + ... + v_n p^{n-2}.$$

Notice $(q_1 - v_1)/p$ is the quotient q_2 of $q_1 \div p$. Hence as before, we have

$$v_2 = \phi_p\left(\frac{q_1 - v_1}{p}\right) = \phi_p(q_2).$$

Continuing in this manner, if we let $q_0 = v$, we get

$$v_i = \phi_p\left(\frac{q_{i-1} - v_{i-1}}{p}\right), \text{ for } 1 \le i \le n-1 \tag{1.18}$$

where the division by p is guaranteed to be an exact integer division. In formula (1.18), it is important to note that the calculation $(q_{i-1} - v_{i-1})/p$ is to be performed in the domain $\mathbb{Z}$ and then the modular mapping ϕ_p is applied. We supplement our definition of the symmetric p-adic integer representation with Example 1.9.

Example 1.9. *Let $v = -284400$. We will find the symmetric p-adic representation of v choosing $p = 103$ using equation (1.18). The p-adic coefficients are*

$$q_0 = v$$

$$v_0 = \phi_p(q_0) = \phi_{103}(-284400) = -17$$

$$v_1 = \phi_p\left(\frac{q_0 - v_0}{p}\right) = \phi_{103}\left(\frac{-284400-(-17)}{103}\right) = \phi_{103}(-1761) = 20$$

$$q_1 = -1761$$

$$v_2 = \phi_p\left(\frac{q_1 - v_1}{p}\right) = \phi_{103}\left(\frac{-1761-20}{103}\right) = \phi_{103}(-27) = -27$$

If we try to compute another coefficient v_3 we find that $q_2 - v_2 = 0$, so we are finished. The symmetric p-adic representation of $v = -284400$ when $p = 103$ is

$$v = -284400 = -17 + 20(103) - 27(103^2).$$

△

The concept of a p-adic representation can be readily extended to polynomials. We can express a polynomial $v(x) \in \mathbb{Z}[x]$ in its *polynomial p-adic representation*, where its general form is

$$v(x) = v_0(x) + v_1(x)p + \cdots + v_{n-1}(x)p^{n-1}$$

where $v_i \in \mathbb{Z}_p[x]$ for $0 \leq i < n$. Formulas (1.17) and (1.18) remain valid when v and v_i for $0 \leq i < n$ are polynomials. We define $\phi_p : \mathbb{Z}[x] \to \mathbb{Z}_p[x]$ by taking the coefficients of a polynomial modulo p.

Example 1.10. *Let $v(x) = 30x^2 - 17x - 29 \in \mathbb{Z}[x]$. We will find the symmetric p-adic representation of v choosing $p = 7$ using equation (1.18). The polynomial p-adic coefficients are:*

$$q_0 = v$$

$$v_0 = \phi_p(q_0) = \phi_7(30x^2 - 17x - 29) = 2x^2 - 3x - 1$$

$$v_1 = \phi_p\left(\frac{v - v_0}{p}\right) = \phi_7\left(\frac{q_0 - v_0}{p}\right) = \phi_7\left(\frac{(30x^2-17x-29)-(2x^2-3x-1)}{7}\right) = \phi_7(4x^2 - 2x - 4)$$
$$= -3x^2 - 2x + 3$$

$$q_1 = 4x^2 - 2x - 4$$

$$v_2 = \phi_p\left(\frac{q_1 - v_1}{p}\right) = \phi_7\left(\frac{(4x^2-2x-4)-(-3x^2-2x+3)}{7}\right) = \phi_7(x^2 - 1) = x^2 - 1$$

If we try to compute another coefficient v_3, we find the $q_2 - v_2 = 0$ so we are finished. The p-adic representation of $v = 30x^2 - 17x - 29$ when $p = 7$ is

$$v(x) = 30x^2 - 17x - 29 = (2x^2 - 3x - 1) + (-3x^2 - 2x + 3)7 + (x^2 - 1)7^2.$$

Note, there exists a positive p-adic representation of $v(x)$ by taking the p-adic coefficients mod 7^2, namely

$$30x^2 - 17x - 29 \equiv (2x^2 + 4x + 6) + (4x^2 + 4x + 2)7 \pmod{7^2}.$$

$\triangle$

We have defined the p-adic representation for polynomials in $\mathbb{Z}[x]$. We can now adapt Hensel's Lemma to factor a polynomial $A(x) \in \mathbb{Z}[x]$ into two factors $u, w \in \mathbb{Z}[x]$. We can recover u and w using their p-adic representation for some prime p. That is, we can represent u and w in the form

$$u = u_0 + u_1 p + u_2 p^2 + \cdots + u_n p^n$$
$$w = w_0 + w_1 p + w_2 p^2 + \cdots + w_n p^n$$

where $u_i = \sum_{j=0}^{m} u_{ij}\, x^j$ and $w_i = \sum_{j=0}^{m} w_{ij}\, x^j$ such that $-\frac{p}{2} < u_{ij}, w_{ij} < \frac{p}{2}$. For simplicity, we define

$$u^{(k)} = u_0 + u_1 p + u_2 p^2 + \cdots + u_{k-1} p^{k-1} \text{ and}$$
$$w^{(k)} = w_0 + w_1 p + w_2 p^2 + \cdots + w_{k-1} p^{k-1}$$

for $k \geq 1$. Given $u_0, w_0 \in \mathbb{Z}_p[x]$, such that $A - u_0 w_0 \equiv 0 \pmod p$ and $\gcd(u_0, w_0)$=1, we wish to find u, w. Using our definition of $u^{(k)}, w^{(k)}$, we can state, using Theorem 1.5, that

$$A - u^{(k)} w^{(k)} \equiv 0 \pmod{p^k}.$$

We need to find $u^{(k+1)} = u^{(k)} + u_k p^k$ and $w^{(k+1)} = w^{(k)} + w_k p^k$ such that

$$\begin{aligned} A - u^{(k+1)} w^{(k+1)} &\equiv 0 \pmod{p^{k+1}} \\ \implies A - (u^{(k)} + u_k p^k)(w^{(k)} + w_k p^k) &\equiv 0 \pmod{p^{k+1}} \\ \implies \underbrace{A - u^{(k)} w^{(k)}}_{1} - \underbrace{u_k p^k w^{(k)}}_{2} - \underbrace{w_k p^k u^{(k)}}_{3} - \underbrace{u_k w_k p^{2k}}_{4} &\equiv 0 \pmod{p^{k+1}}. \end{aligned}$$

Now, we can consider 1, 2, 3, 4 separately mod p^{k+1}.

1 We know that $A = uw$. It follows that

$$
\begin{aligned}
A - u^{(k)}w^{(k)} &= A - (u_0 + u_1p + ... + u_{k-1}p^{k-1})(w_0 + w_1p + ... + w_{k-1}p^{k-1}) \\
&= A - u_0w_0 - (u_0w_1 + u_1w_0)p - ... - (u_0w_{k-1} + u_1w_{k-2} + ... + u_{k-1}w_0)p^{k-1} - (...)p^k
\end{aligned}
$$

As $u^{(k)}w^{(k)}$ generates the complete p-adic representation of A for the first k coefficients, it is implied that $A - u^{(k)}w^{(k)}$ contains only terms that are multiples of p^k. We will define a new quantity, the **error** e_k at iteration k, by

$$e_k = A - u^{(k)}w^{(k)}$$

which represents the difference between A and the incomplete factorization of $u^{(k)}w^{(k)}$. As every term in e_k is a multiple of at least p^k, we need only consider the coefficients which are factors of exactly p^k modulo p^{k+1}. So, if we divide e_k by p^k, we define the quantity

$$c_k = \frac{e_k}{p^k}$$

for some $c_k \in \mathbb{Z}[x]$.

2 Consider the expansion of $w^{(k)}$ divided by p^k

$$\frac{u_kp^kw^{(k)}}{p^k} = u_k(w_0 + w_1p + ... + w_{k-1}p^{k-1}) \equiv u_kw_0 \pmod p$$

3 Using the same logic as used in 2,

$$\frac{w_kp^ku^{(k)}}{p^k} = w_ku^{(k)} \equiv u_0w_k \pmod p.$$

4 Every term is a multiple of p^{2k} and $k \geq 1$, so

$$\frac{u_kw_kp^{2k}}{p^k} \equiv 0 \pmod p.$$

If we put together our results from 1, 2, 3, and 4, we conclude

$$
\begin{aligned}
A - (u^{(k+1)}w^{(k+1)}) \equiv 0 \pmod{p^{k+1}} &\implies \frac{e_k - u_kp^kw^{(k)} - w_kp^ku^{(k)} - u_kw_kp^{2k}}{p^k} \equiv 0 \pmod p \\
&\implies \frac{e_k}{p^k} - u_kw_0 - w_ku_0 \equiv 0 \pmod p \\
&\implies u_kw_0 + u_0w_k = c_k \in \mathbb{Z}_p[x].
\end{aligned}
$$

Now we must find u_k and w_k. We have a polynomial Diophantine equation of the form

$$\sigma w_0 + \tau u_0 = c_k$$

for a known polynomial $c_k \in \mathbb{Z}[x]$. Since $\gcd(u_0, w_0) = 1$ in $\mathbb{Z}_p[x]$, we can apply Theorem 1.3 and find a unique solution for $\sigma, \tau \in \mathbb{Z}_p[x]$ with $\deg(\sigma, x) < \deg(u_0, x)$. We now set

$$u^{(k+1)} = u^{(k)} + \sigma p^k \quad \text{and} \quad w^{(k+1)} = w^{(k)} + \tau p^k.$$

The Hensel construction provides a method for lifting a factorization modulo p up to a factorization modulo p^k for $k \geq 1$. However, the Hensel construction may not lead to a factorization over $\mathbb{Z}[x]$. While Corollary 1.7 guarantees a unique construction if the relatively prime factors $u^{(1)}$ and $w^{(1)}$ are monic, this is not true if the initial factors are not monic. Hensel construction can find polynomials that satisfy the conditions of Theorem 1.5, but as the construction is not unique, the polynomials calculated might not be the factorization of A. Therefore, we must discuss a stopping condition for Hensel Lifting. The stopping condition will occur when k is large enough that p^k is larger than any possible integer coefficient of factors u or w. Specifically, let k be large enough so that $p^k > 2B$ where B is an integer which bounds the magnitudes of all integer coefficients appearing in $A(x)$ and in any of its possible factors with the particular degrees $\deg(u^{(1)}, x)$ and $\deg(w^{(1)}, x)$. For an upper bound on the size of the coefficients we will use Mignotte's bound. We present Mignotte's bound as Theorem 1.11. A proof of Mignotte's bound can be found in [6]. The bound was originally presented in [14].

Theorem 1.11. (*Mignotte's bound*) *Suppose that* $f, g, h \in \mathbb{Z}[x]$ *have degrees* $\deg(f, x) = n \geq 1, \deg(g, x) = m$*, and* $\deg(h, x) = k$*, and that* gh *divides* f *(in* $\mathbb{Z}[x]$*). For a polynomial* $A \in \mathbb{Z}[x]$*, let* $||A||_\infty$ *be the magnitude of the coefficient of* A *with the largest magnitude. Then*

$$\begin{array}{ll} (1) & ||g||_\infty ||h||_\infty \leq \sqrt{n+1}\; 2^{m+k} ||f||_\infty, \\ (2) & ||h||_\infty \leq \sqrt{n+1}\; 2^k ||f||_\infty. \end{array}$$

Proof. See Corollary 6.33 in [6]. □

The basic algorithm for Hensel Lifting a factorization in $\mathbb{Z}_p[x]$ up to a factorization in $\mathbb{Z}[x]$ is presented as Algorithm 1. In the monic case, Algorithm 1 corresponds precisely to the Hensel construction presented above, since by Corollary 1.7 the factors at each step of the lifting process are uniquely determined in the monic case. However, in the non-monic case, the non-uniqueness of the factors modulo p^k leads to the "leading coefficient problem" to be discussed shortly, and as we shall see this accounts for the additional conditions and operations appearing in Algorithm 1. For the moment, Algorithm 1 may be understood for the monic case if we simply ignore the stated conditions (other than the conditions appearing in Hensel's lemma), ignore Steps 3-9 and initialize

$u \leftarrow u_0$ and $w \leftarrow w_0$. Note that no adjustment to u and w is needed in Steps 14 and 15. We will define the terms monic, lcoeff, content, and primitive when we discuss the "leading coefficient problem".

Algorithm 1: Hensel Lifting in 2 factors

---

1 **Input**: prime p, primitive $A \in \mathbb{Z}[x]$ and $u_0, w_0 \in \mathbb{Z}_p[x]$ satisfying $(i) A \equiv u_0 w_0 \pmod p$ and $(ii) \gcd(u_0, w_0) = 1$, and B, an upper bound for the magnitude of the integer coefficients of u, w.

2 **Output**: $u, w \in \mathbb{Z}[x]$ such that $A = uw$ or FAIL.

3 $\alpha \leftarrow \text{lcoeff}(A, x)$;

4 $A \leftarrow \alpha A$;

5 $du \leftarrow \deg(u_0, x); dw \leftarrow \deg(w_0, x)$;

6 $u_0 \leftarrow \alpha \cdot \text{ monic}(u_0) \mod p$;

7 $w_0 \leftarrow \alpha \cdot \text{ monic}(w_0) \mod p$;

8 $u \leftarrow u_0 - \text{ lcoeff}(u_0, x) x^{du} + \alpha x^{du}$;

9 $w \leftarrow w_0 - \text{ lcoeff}(w_0, x) x^{du} + \alpha x^{dw}$;

10 $k \leftarrow 1$;

11 **while** $p^k < 2B$ **do**

12     $e_k \leftarrow A - uw$;

13   **if** $e_k = 0$ **then**

14         $u \leftarrow u/\text{content}(u)$; $w \leftarrow w/\text{content}(w)$;

15   **return**(u, w);

16   **end**

17     $c_k \leftarrow e_k / p^k$;

18   Solve $\sigma w_0 + \tau u_0 = c_k$ for $\sigma, \tau \in \mathbb{Z}_p[x]$;

19     $u \leftarrow u + \sigma p^k$; $w \leftarrow w + \tau p^k$;

20     $k \leftarrow k + 1$;

21 **end**

22 **return** FAIL **end**

Example 1.12. *We will apply Algorithm 1 to factor the following monic polynomial over the integers:*

$$A(x) = x^4 + 57x^3 - 73493x^2 + 74631x - 18860 \in \mathbb{Z}[x].$$

Choosing $p = 5$ and applying the modular homomorphism ϕ_5 to A yields

$$\phi_5(A) = x^4 + 2x^3 + 2x^2 + x \in \mathbb{Z}_5[x].$$

The unique monic factorization in $\mathbb{Z}_5[x]$ of this polynomial is

$$\phi_5(A) = (x^2 + x)(x^2 + x + 1).$$

We therefore define

$$u^{(1)}(x) = x^2 + x, \quad \text{and} \quad w^{(1)}(x) = x^2 + x + 1.$$

Since $u^{(1)}$ and $w^{(1)}$ are relatively prime in $\mathbb{Z}_5[x]$, the Hensel construction may be applied. Applying Algorithm 1 in the form noted above for the monic case, we first perform the initializations in Steps 8,9, and 12 which yields

$$u(x) = x^2 + x,\ w(x) = x^2 + x + 1,\ \text{and}\ e(x) = 55x^3 - 73496x^2 + 74630x - 18860.$$

Steps 17-19 then apply the Hensel construction precisely as outlined in the proof of Hensel's Lemma. The sequence of values computed for $e, c_k, \sigma_k, \tau_k, u$, and w in Step 19 are as follows.

k	e_k	c_k	σ_k	τ_k	u	w
1	$55x^3 - 73495x^2 + 74630x - 18860$	$x^3 + x^2 + x - 2$	$x - 2$	2	$x^2 + 6x - 10$	$x^2 + x + 11$
2	$50x^3 - 73500x^2 + 74575x - 18750$	$2x^3 - 2x$	0	$2x - 2$	$x^2 + 6x - 10$	$x^2 + 51x - 39$
3	$-73750x^2 + 75375x - 19250$	$-2x + 1$	$2x + 1$	$2x - 1$	$x^2 - 244x + 115$	$x^2 + 301x - 164$
4	0	-	-	-	-	-

Table 1.2: Value of variables after Step 19

Note that at the end of each iteration k, $e_k(x)$ is divisible by 5^{k+1} as required at the beginning of the next iteration. The iteration terminates with $u(x) = x^2 - 244x + 115$ and $w(x) = x^2 + 301x - 164$. We therefore have the factorization

$$x^4 + 57x^3 - 73493x^2 + 74631x - 18860 = (x^2 - 244x + 115)(x^2 + 301x - 164).$$

A variation of Hensel Lifting exists that lifts A, u and w in the positive range. In this case, as the loop terminated at $k = 3$, we could put A, u, and w into the positive range mod 5^3. In this case, we would have the factorization of $A \equiv (x^2 + 6x + 115)(x^2 + 51x + 86) \pmod{5^3}$.

△

The Leading Coefficient Problem

The Hensel construction provides a method for lifting a factorization modulo p up to a factorization modulo p^k for any integer $k \geq 1$. However, if the monic polynomial $A(x) \in \mathbb{Z}[x]$ has the factorization

$$A \equiv u^{(1)} w^{(1)} \pmod{p}$$

where $u^{(1)}, w^{(1)} \in \mathbb{Z}_p[x]$ are relatively prime monic polynomials and if there exists a factorization over the integers $A = uw$ such that

$$u \equiv u^{(1)} \pmod{p} \quad \text{and} \quad w \equiv w^{(1)} \pmod{p}$$

then the Hensel construction must obtain this factorization by Corollary 1.7. Notice that subtracting $u^{(1)}w^{(1)}$ from A removes the leading monic term from A resulting in a polynomial of degree at most $\deg(A, x) - 1$. As the leading terms always cancel in the monic case, when the polynomial Diophantine equation

$$\sigma w^{(1)} + \tau u^{(1)} = c_k$$

is solved for $\sigma, \tau \in \mathbb{Z}_p[x]$ where $\deg(\sigma, x) < \deg(u^{(1)}, x)$ and $\deg(c_k, x) < \deg(A, x)$, then $\deg(u^{(1)}w^{(1)}, x) = \deg(A, x)$ and the leading coefficients of $u^{(k)}$ and $w^{(k)}$ never change. This is fine when A is monic, but is a problem is A is non-monic as the values of the leading coefficients will never change from their initial values. We must discuss how to handle the case when A, and therefore at least one of the factors, is non-monic. As our algorithm, introduced in Section 3.1, is intended to factor bivariate polynomials that are monic in x, we will not provide the complete explanation of how to solve the leading coefficient problem. A complete explanation can be read in pages 237-250 of [7].

We will assume the leading coefficient of $A(x) \in \mathbb{Z}[x]$ is $\alpha \neq 1$. We want to find some $\beta, \gamma \in \mathbb{Z}$ such that $\text{lcoeff}(A) = \alpha = \beta\gamma = \text{lcoeff}(u)\text{lcoeff}(w)$. We have $u^{(1)}, w^{(1)}$ such that $\gcd(u^{(1)}, w^{(1)}) = 1$ and $A - u^{(1)}w^{(1)} \equiv 0 \pmod{p}$. Since we are working over the field $\mathbb{Z}_p$, each of the polynomials can be written as a product of their leading coefficients and their monic forms. In other words, if the leading coefficients of $u^{(1)}$ and $w^{(1)}$ are β', γ' respectively then $\alpha\,\text{monic}(A) \equiv \beta'\,\text{monic}(u^{(1)}) \cdot \gamma'\,\text{monic}(w^{(1)}) \pmod{p}$.

Consider the following factorization of A:

$$\alpha\,\text{monic}(A) = \beta\,\text{monic}(u) \times \gamma\,\text{monic}(w). \tag{1.19}$$

Now, if we multiply both sides of (1.19) by α we get

$$\begin{aligned} \alpha^2\,\text{monic}(A) &= \alpha(\beta\text{monic}(u) \times \gamma\text{monic}(w)) \\ &= \beta\gamma\,(\beta\,\text{monic}(u) \times \gamma\,\text{monic}(w)) \\ &= \beta\gamma\,\text{monic}(u) \times \beta\gamma\,\text{monic}(w) \\ &= \alpha\,\text{monic}(u) \times \alpha\,\text{monic}(w). \end{aligned} \tag{1.20}$$

The first step to solving the leading coefficient problem is to multiply the monic forms of $u^{(1)}$ and $w^{(1)}$ by α, then when we initialize u and w in Steps 8-9 of Algorithm 1, we can simply set the leading coefficients to both be α. In this way, when we calculate $A-uw$ in Step 12, the leading terms cancel, $\deg(e_k, x) = \deg(c_k, x < \deg(A, x)$, so we are able to apply Theorem 1.3 to the polynomial Diophantine equation to solve for σ and τ and update u and w.

The last thing to notice is if we do find a factorization $A = uw$, then there will be an extra factor of α, the leading coefficient of A, split between u and w. To remove this, we need to find the *content* of u and w. The content of a polynomial $u \in \mathbb{Z}[x]$ is the gcd of its coefficients. We then divide the coefficients of u and w by their respective contents. This corresponds to Steps 14 and 15 in Algorithm 1 and results in correct factorization of A.

Example 1.13. *We will apply the entirety of Algorithm 1 to factor the following non-monic polynomial over the integers:*

$$A(x) = 48x^4 - 22x^3 + 47x^2 + 144.$$

The input of the algorithm is the primitive polynomial A, the prime is $p = 7$ (note that p does not divide the leading coefficient of A), and the two relatively prime modulo p factors of A are $u^{(1)}(x) = x^2 - 3x + 2$ and $w^{(1)}(x) = -x^2 + 3x + 2$.

We begin by finding the monic representation of $u^{(1)}$ and $w^{(1)}$ in $\mathbb{Z}_p[x]$. We then multiply $u^{(1)}$ and $w^{(1)}$ by α. So,

$$\alpha A = 2304x^4 - 1056x^3 + 2256x^2 + 6912$$

$$\begin{aligned} u^{(1)} &= \alpha \cdot \text{monic}(u^{(1)}) &&\equiv -x^2 + 3x - 2 \pmod 7, \text{ and} \\ w^{(1)} &= \alpha \cdot \text{monic}(w^{(1)}) &&\equiv -x^2 + 3x + 2 \pmod 7. \end{aligned}$$

We then can replace the leading term of $u^{(1)}$ and $w^{(1)}$ with $\alpha = 48$ so that when we initialize u and w we get

$$u = 48x^2 + 3x - 2 \text{ and } w = 48x^2 + 3x + 2.$$

We can then calculate the error e_1 and c_1 as

$$e_1 = A - uw = -1344x^3 + 2247x^2 + 6916, \text{ and}$$

$$c_1 = \frac{e_1}{p} = -192x^3 + 321x^2 + 988.$$

We can then solve the Diophantine equation $\sigma w^{(1)} + \tau u^{(1)} = c_1$ for $\sigma, \tau \in \mathbb{Z}_7$. We get $\sigma_1 = x$ and $\tau_1 = 2x + 3$. From here, we can update the values of u and w to get

$$u = 48x^2 + 10x - 2 \ \ \textit{and} \ \ w = 48x^2 + 17x + 23.$$

We can then repeat these steps to solve for u and w. The sequence of values computed for $e, c_k, \sigma_k, \tau_k, u$, and w in Step 19 are as follows.

k	e_k	c_k	σ_k	τ_k	u	w
1	$-1344x^3 + 2247x^2 + 6916$	$-192x^3 + 321x^2 + 988$	x	$2x + 3$	$48x^2 + 10x - 2$	$48x^2 + 17x + 23$
2	$-2352x^3 + 1078x^2 - 196x + 6958$	$-48x^3 + 22x^2 - 4x + 142$	$-2x + 2$	$x + 1$	$48x^2 - 88x + 96$	$48x^2 + 66x + 72$
3	0	-	-	-	-	-

Table 1.3: Value of variables after Step 19

Finally, in Steps 13-17, we calculate $\delta =$ content(u), and find the factorization of A:

$$\begin{array}{lcl} \delta & = & \text{content}(48x^2 - 88x + 96) = 8, \\ u(x) & = & \frac{48x^2 - 88x + 96}{8} = 6x^2 - 11x + 12, \\ w(x) & = & \frac{48x^2 + 66x + 72}{48/8} = 8x^2 + 11x + 12. \end{array}$$

We have found the factorization of A, namely

$$A(x) = 48x^4 - 22x^3 + 47x^2 + 144 = (6x^2 - 11x + 12)(8x^2 + 11x + 12).$$

$\triangle$

1.1.4 The cost of Hensel Lifting mod p^m

The cost of Algorithm 1 is dominated by the cost of the kth step of Hensel Lifting where we compute the error e_k in Step 12 as

$$e_k = A - u^{(k)} w^{(k)}.$$

Suppose $\deg(A, x) = d$ and $\deg(u, x) = \deg(w, x) = d/2$ which maximizes the cost of this step. Here $u^{(k)}$ and $w^{(k)}$ have degree $d/2$ with coefficients in the range $(-\frac{p^k}{2}, \frac{p^k}{2})$. Multiplying $u^{(k)} \times w^{(k)}$ costs $O(\frac{d}{2}^2 k^2)$ if we use classical multiplication in $\mathbb{Z}$ and $\mathbb{Z}[x]$. Therefore, the cost of lifting $u^{(1)}$ and $w^{(1)}$ to p^m is

$$\sum_{k=1}^{m} O(d^2 k^2) = O\left(d^2 \sum_{k=1}^{m} k^2\right) = O\left(d^2 \frac{m(m+1)(2m+1)}{6}\right) = O(d^2 m^3).$$

Algorithm 1 performs $O(d^2\, m^3)$ multiplications in $\mathbb{Z}_p$. It should be mentioned that Miola and Yun [15] reduced the complexity of Linear Hensel Lifting in $\mathbb{Z}[x]$ to $O(d^2 m^2)$ by avoiding the re-computation of e_{k-1} at each lifting step. They calculated e_k/p^k using

$$
\begin{aligned}
\frac{e_k}{p^k} &= \frac{A - u^{(k)}w^{(k)}}{p^k} \\
&= \frac{A - (u^{(k-1)} + u_{k-1}p^{k-1})(w^{(k-1)} + w_{k-1}p^{k-1})}{p^k} \\
&= \frac{A - (u^{(k-1)}w^{(k-1)} + w_{k-1}u^{(k-1)}p^{k-1} + u_{k-1}w^{(k-1)}p^{k-1} + u_{k-1}w_{k-1}p^{2(k-1)})}{p^k} \\
&= \frac{e_{k-1} - w_{k-1}u^{(k-1)} - u_{k-1}w^{(k-1)} + u_{k-1}w_{k-1}p^{k-1}}{p}
\end{aligned}
$$

to reduce the number of multiplications done when calculating e_k. Miola and Yun's method does not generalize to $n > 2$ factors. As Bernardin uses a different method to compute c_k in each lifting step, Miola and Yun's optimization will not be used.

1.1.5 Quadratic Hensel Lifting in $\mathbb{Z}[x]$

Zassenhaus [25] in 1969 was the first to propose the application of Hensel's Lemma to the problem of polynomial factorization over the integers and he proposed the use of a quadratic p-adic Newton iteration. This quadratic iteration is usually referred to as the *Zassenhaus construction* and it computes a sequence of factors modulo p^{2^k}, for $k = 1, 2, 3, ...$ However, the additional cost of a quadratic iteration, in comparison with the linear algorithm previously described, may outweigh the advantage of fewer iteration steps. For example, in each iteration step of the quadratic Zassenhaus construction one must solve a polynomial Diophantine equation of the form

$$\sigma(x)u^{(k)}(x) + \tau(x)w^{(k)}(x) \equiv c(x) \pmod{p^{2^{k-1}}} \tag{1.21}$$

for $\sigma, \tau \in \mathbb{Z}_{p^{2^{k-1}}}[x]$. The corresponding computation in the linear Hensel construction is to solve the same polynomial Diophantine equation modulo p for $\sigma, \tau \in \mathbb{Z}_p[x]$. The latter computation is simpler because it is performed in the smaller domain $\mathbb{Z}_p[x]$, and another level of efficiency arises because $u^{(k)}$ and $w^{(k)}$ in (1.21) can be replaced by the fixed polynomial $u^{(1)}$ and $w^{(1)}$ in the linear Hensel case. A detailed comparison of these two p-adic constructions was carried out of Miola and Yun [15] in 1974 and their analysis showed that the computational cost of the quadratic Zassenhaus construction is higher than that of the linear Hensel construction for achieving the same p-adic order of approximation.

1.2 Hensel Lifting in $\mathbb{F}_p[x, y]$

In the previous section, we presented the Linear Hensel Lifting algorithm for polynomials in $\mathbb{Z}[x]$ for two factors. We had to lift two polynomials from $\mathbb{Z}_p$ to $\mathbb{Z}$. We now have to lift n univariate polynomials in $\mathbb{F}_p[x]$ to bivariate polynomials in $\mathbb{F}_p[x, y]$. We present an algorithm that is capable of performing Linear Hensel Lifting on polynomials in $\mathbb{F}_p[x, y]$ for $n \geq 2$ factors and some prime p. This can be done by modifying the univariate Hensel Lift algorithm for two factors from Section

1.1, Algorithm 1, to work for bivariate polynomials over some finite field $\mathbb{F}_p$. In this section, we will describe all the changes between Algorithm 1 and the bivariate Hensel Lifting algorithm in $\mathbb{F}_p[x, y]$ for $n \geq 2$ factors.

Let $A(x, y) \in \mathbb{F}_p[x, y]$ with $d_x = \deg(A, x) > 0$ and $d_y = \deg(A, y) > 0$. We will assume $A(x, y)$ is monic in x. Unlike the previous section, we will assume that we are factoring A into $n \geq 2$ factors $f_1, f_2, ..., f_n \in \mathbb{F}_p[x, y]$ such that $A = f_1 \times f_2 \times ... \times f_n$. The non-monic case will not be discussed.

The first change we will discuss is how we represent each of the n factors. Previously, we represented the factors as a p-adic representation around some prime p. We wrote the factor u as

$$u = \sum_{i=0}^{\ell-1} u_i p^i$$

where $u_i \in \mathbb{Z}_p[x]$ for $0 \leq i < \ell$. For $\mathbb{F}_p[x, y]$, we represent the factors as a power series around $(y - \alpha)$ for some $\alpha \in \mathbb{F}_p$. In other words, we will define each factor f_i as

$$f_i = \sum_{j=0}^{m-1} f_{i,j} (y - \alpha)^j$$

where $f_{i,j} \in \mathbb{F}_p[x]$ for $0 \leq j < m$. Whereas for the univariate Hensel Lifting algorithm we lifted a polynomial from $\mathbb{Z}_p[x]$ to $\mathbb{Z}[x]$, we are now attempting to lift a univariate polynomial in $\mathbb{F}_p[x]$ to a bivariate polynomial in $\mathbb{F}_p[x, y]$.

Suppose that $A(x, y)$ is square-free in $\mathbb{F}_p[x, y]$ and we are given the factorization of $A(x, \alpha) = \prod_{i=1}^{n} f_{i,0}$ where $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ are pairwise relatively prime. We pick α such that $A(x, \alpha)$ is square-free. Now

$$\begin{aligned}
A(x, \alpha) \text{ is square-free} &\iff \gcd\left(A(x, \alpha), \frac{\partial A}{\partial x}(x, \alpha)\right) = 1 \text{ by Theorem 8.1 of [7]} \\
&\iff \operatorname{res}\left(A(x, \alpha), \frac{\partial A}{\partial x}(x, \alpha), x\right) \neq 0 \text{ by Proposition 8 in Ch. 3.5 of [4]} \\
&\iff \operatorname{res}\left(A(x, y), \frac{\partial A}{\partial x}(x, y), x\right)(y = \alpha) \neq 0 \text{ as } A \text{ is monic in } x
\end{aligned}$$

where $\operatorname{res}(f, g, x)$ is the Sylvester resultant. Now, $\operatorname{res}\left(A, \frac{\partial A}{\partial x}(x, y), x\right)$ is a polynomial in $\mathbb{F}_p[y]$ of finite degree, so there are only finitely many roots.

We are aiming to construct monic, bivariate polynomials $f_1^{(k)}, f_2^{(k)}, ..., f_n^{(k)} \in \mathbb{F}_p[x, y]$ where

$$f_i^{(k)} = f_i^{(k-1)} + f_{i,k-1}(y-\alpha)^{(k-1)} \text{ and } f_i^{(1)} = f_{i,0}$$

such that

$$\begin{array}{ll} (1) & \forall i : f_i^{(k)} \equiv f_{i,0} \pmod{y-\alpha}, \text{ and} \\ (2) & A \equiv \prod_i f_i^{(k)} \pmod{(y-\alpha)^k}. \end{array}$$

If k is sufficiently large, the $f_i^{(k)}$ obtained can be used to compute a factorization of A over $\mathbb{F}_p$.

The single most noticeable change when going from 2 to n factors is the Diophantine equation that we must solve. The method for solving the Diophantine equation introduced in Section 1.1.1 only can only find the lifting coefficients for exactly two factors. We must obtain a new Diophantine equation that will allow us to solve for the n lifting coefficients of that factor in the bivariate Hensel Lifting algorithm. Using the condition $A - \prod_{i=1}^n f_i^{(k+1)} \equiv 0 \pmod{(y-\alpha)^{(k+1)}}$, we know

$$\begin{aligned} A - \prod_{i=1}^n f_i^{(k+1)} &\equiv 0 \pmod{(y-\alpha)^{(k+1)}} \\ \implies A - \prod_{i=1}^n (f_i^{(k)} + f_{i,k}(y-\alpha)^k) &\equiv 0 \pmod{(y-\alpha)^{(k+1)}} \\ \implies \underbrace{A - \prod_{i=1}^n f_i^{(k)}}_{1} - \underbrace{\sum_{i=1}^n \frac{\prod_{j=1}^n f_j^{(k)}}{f_i^{(k)}} f_{i,k}(y-\alpha)^k}_{2} + \underbrace{(\dots)(y-\alpha)^{2k}}_{3} &\equiv 0 \pmod{(y-\alpha)^{(k+1)}}. \end{aligned} \tag{1.22}$$

Now we can consider 1, 2, and 3 separately mod $(y-\alpha)^{k+1}$.

1 We know that $A = f_1 f_2 ... f_n$. This implies that $A - \prod_{i=1}^n f_i^{(k)}$ contains only terms that are multiples of $(y-\alpha)^j$ for $j \geq k$. We will define the error e_k at iteration k for Hensel Lifting with n factors as

$$e_k = A - \prod_{i=1}^n f_i^{(k)}$$

which represents the difference between A and the possibly incomplete factorization of $\prod_{i=1}^n f_i^{(k)}$. As every term in e_k is a multiple of at least $(y-\alpha)^k$ and we are seeking the terms modulo $(y-\alpha)^{k+1}$, if we divide e_k by $(y-\alpha)^k$, we get the polynomial

$$c_k = \frac{e_k}{(y-\alpha)^k}$$

for some c_k. The terms we desire can be obtained by taking c_k modulo $(y-\alpha)$.

2 Consider the expansion of $\frac{\prod_{j=1}^n f_j^{(k)}}{f_i^{(k)}} f_{i,k}(y-\alpha)^k$ divided by $(y-\alpha)^k$. Using $f_i^{(k)} = \sum_{j=0}^{k-1} f_{i,j}(y-\alpha)^k$ we find

$$\frac{\sum_{i=1}^n \frac{\prod_{j=1}^n f_j^{(k)}}{f_i^{(k)}} f_{i,k}(y-\alpha)^k}{(y-\alpha)^k} = \sum_{i=1}^n \frac{\prod_{j=1}^n f_j^{(k)}}{f_i^{(k)}} f_{i,k} = \sum_{i=1}^n f_{i,k} \prod_{\substack{j=1\\ j\neq i}}^n f_j^{(k)} \equiv \sum_{i=1}^n f_{i,k} \prod_{\substack{j=1\\ j\neq i}}^n f_{j,0} \pmod{(y-\alpha)}.$$

3 Easiest of all, as the remaining terms are all multiples of at least $(y-\alpha)^{2k}$, they are congruent to $0 \pmod{y-\alpha}$ after dividing by $(y-\alpha)^k$.

Combining our results from 1, 2, and 3, we have

$$\begin{aligned}
&\tfrac{A-\prod_{i=1}^n f_i^{(k+1)}}{(y-\alpha)^k} \equiv 0 \pmod{(y-\alpha)}\\
&\implies \frac{e_k - \sum_{i=1}^n \frac{\prod_{j=1}^n f_j^{(k)}}{f_i^{(k)}} f_{i,k}(y-\alpha)^k - (\dots)(y-\alpha)^{2k}}{(y-\alpha)^k} \pmod{(y-\alpha)^k} \equiv 0 \pmod{(y-\alpha)}\\
&\implies \tfrac{e_k}{(y-\alpha)^k} - \sum_{i=1}^n \frac{\prod_{j=1}^n f_j^{(k)}}{f_i^{(k)}} f_{i,k} \equiv 0 \pmod{(y-\alpha)}\\
&\implies c_k - \sum_{i=1}^n \frac{\prod_{j=1}^n f_{j,0}}{f_{i,0}} f_{i,k} \equiv 0 \pmod{(y-\alpha)}.
\end{aligned}$$

As we have shown $c_k = \sum_{i=1}^n \frac{\prod_{j=1}^n f_{j,0}}{f_{i,0}} f_{i,k}$ in $\mathbb{F}_p[x]$. We have derived the multi-Diophantine equation we must solve to obtain the lifting coefficients for n factors. We need to solve the polynomial Diophantine equation

$$c_k \equiv \prod_{i=1}^n \sigma_i \frac{\prod_{j=1}^n f_{j,0}}{f_{i,0}} \pmod{(y-\alpha)} \tag{1.23}$$

for $\sigma_1, ..., \sigma_n \in \mathbb{F}_p[x]$ such that $\deg(\sigma_i, x) < \deg(f_{i,0}, x)$ for $1 \leq i \leq n$. There are several ways to solve (1.23). We will provide an efficient method to solve it later in Section 2.2.2. We show that our method for solving (1.23) uses $O(d_x^2)$ arithmetic operations in $\mathbb{F}_p$ where $d_x = \deg(A, x)$.

The remaining changes are trivial. We must now initialize and update n factors instead of two. The stopping criteria have also changed. In $\mathbb{Z}[x]$, we used to have to lift until p^k exceeds the upper bound for the magnitude of the coefficients of the factors, but now we must perform the main loop of Hensel lifting until $\sum_{i=1}^{n} \deg(f_i^{(k)}, y) \geq \deg(A, y)$.

We compute the error as $e_k = A - \prod_{i=1}^{n} f_i^{(k)}$ and the polynomial $c_k = e_k/(y-\alpha)^k$. Miola and Yun in [15] describe a way to improve how the error is calculated and in doing so reduce the complexity by a factor of d_y. We demonstrated their method in Section 1.1.4 and it reduced the complexity of Algorithm 1 by a factor of m, where m refers to the power of p we lift the factors $u^{(1)}$ and $w^{(1)}$ to during the analysis of the classical Hensel Lifting algorithm in Section 1.1.4. If we follow Miola and Yun's technique, we could calculate c_k as

$$
\begin{aligned}
c_k = \frac{e_k}{(y-\alpha)^k} &= \frac{A - \prod_{i=1}^{n} f_i^{(k)}}{(y-\alpha)^{(k)}} \\
&= \frac{A - \left(f_1^{(k-1)} + f_{1,k-1}(y-\alpha)^{k-1}\right)\left(f_2^{(k-1)} + f_{2,k-1}(y-\alpha)^{k-1}\right) \dots \left(f_n^{(k-1)} + f_{n,k-1}(y-\alpha)^{k-1}\right)}{(y-\alpha)^k} \\
&= \frac{A - \left(\prod_{i=1}^{n} f_i^{(k-1)} + \sum_{i=1}^{n} f_{i,k-1} \frac{\prod_{j=1}^{n} f_j^{k-1}}{f_i^{(k-1)}}(y-\alpha)^{k-1} + \dots + \sum_{i=1}^{n} f_{i,k-1}(y-\alpha)^{n(k-1)}\right)}{(y-\alpha)^k} \\
&= \frac{\frac{e_{k-1}}{(y-\alpha)^{k-1}} + \sum_{i=1}^{n} f_{i,k-1} \frac{\prod_{j=1}^{n} f_j^{(k-1)}}{f_i^{(k-1)}} + \dots + \sum_{i=1}^{n} f_{i,k-1}(y-\alpha)^{(n-1)(k-1)}}{(y-\alpha)}
\end{aligned}
$$

to avoid recomputing e_{k-1}. However, when $n > 2$, we are forced to multiply bivariate polynomials, which increases the overall number of arithmetic operations. Therefore, we will not use Miola and Yun's improvement technique. Instead we calculate the error e_k using the same method we used in Algorithm 1 of Section 1.1.

We present the bivariate Hensel Lifting algorithm as Algorithm 2 below. The order terms on the right count arithmetic operations in $\mathbb{F}_p$.

By far, the most expensive step in Algorithm 2 is the computation of the error in Step 6. First, we consider calculating the product $\prod_{i=1}^{n} f_i^{(k)}$. Since $\deg(f_i^{(k)}, x) < d_x$ and $\deg(f_i^{(k)}, y) = k-1 < d_y$, this implies that calculating the product results in $O(k^2\, n\, d_x^2)$ arithmetic operation in $\mathbb{F}_p$. The overall cost for the products is $\sum_{k=1}^{d_y} O(k^2\, n\, d_x^2) = O(n\, d_x^2\, d_y^3)$.

Algorithm 2: Classical Linear Bivariate Hensel Lifting for $\mathbb{F}_p[x, y]$: Monic Case.

Input: prime p, $\alpha \in \mathbb{F}_p$, $A \in \mathbb{F}_p[x, y]$ and $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ satisfying
$(i) A, f_{1,0}, f_{2,0}, ..., f_{n,0}$ are monic in x, $(ii) A(x, y = \alpha) = f_{1,0} f_{2,0} ... f_{n,0}$ and
$(iii) \gcd(f_{i,0}, f_{j,0}) = 1$ for $i \neq j$.

Output: $f_1, f_2, ..., f_n \in \mathbb{F}_p[x, y]$ such that $A = f_1\, f_2 \ldots f_n$ or FAIL.

$d_x \leftarrow \deg(A, x)$; $d_y \leftarrow \deg(A, y)$;

for $i = 1$ to n **do** $f_i \leftarrow f_{i,0}$; $df_i \leftarrow 0$; **end do**

for $k = 1$ to d_y **while** $\sum_{i=1}^{n} df_i \leq d_y$ **do**

  $e_k \leftarrow A - \prod_{i=1}^{n} f_i$; .... $O(k^2\, n\, d_x^2)$

  $c_k \leftarrow e_k / (y - \alpha)^k$; .... $O(k\, d_x\, d_y)$

  $c_k \leftarrow c_k \pmod{(y - \alpha)}$; .... $O(d_x d_y)$

  if $c_k \neq 0$ **then**

    Solve $\sum_{i=1}^{n} \sigma_i \frac{f_{1,0} f_{2,0} \ldots f_{n,0}}{f_{i,0}} = c_k$ in $\mathbb{F}_p[x]$ for $\sigma_1, \sigma_2, ..., \sigma_n \in \mathbb{F}_p[x]$; .... $O(d_x^2)$

    for $i = 1$ to n **do**

      $f_{i,k} \leftarrow \sigma_i$;

      $f_i \leftarrow f_i + f_{i,k}(y - \alpha)^k$; .... $O(k d_x)$

      $df_i \leftarrow \deg(f_i, y)$;

    end

  end

end

if $error = 0$ **then return** $f_1, f_2, ..., f_n$ **else return** FAIL **end if**

1.3 Bernardin's algorithm

Suppose we are given the initial factorization of $A(x, \alpha) = f_{1,0}(x)f_{2,0}(x)...f_{n,0}(x)$ for some $\alpha \in \mathbb{F}_p$ such that $\gcd(f_{i,0}, f_{j,0}) = 1$ in $\mathbb{F}_p[x]$ for $i \neq j$. We are looking for a factorization of $A = f_1 \times f_2 \times \cdots \times f_n$ with $f_{i,0} = f_i(x, \alpha)$ for $1 \leq i \leq n$. As we described in the previous section, we can define the k'th order approximation of each factor to be

$$f_i^{(k)} = \sum_{j=0}^{k-1} f_{i,j}(x)(y-\alpha)^j \text{ for } 1 \leq i \leq n$$

with $f_{i,j} \in \mathbb{F}_p[x]$ for $0 \leq j \leq k-1$ such that $A - \prod_{i=1}^n f_i^{(k)} \equiv 0 \mod (y-\alpha)^k$.

In 1998, Laurent Bernardin [2] published a paper describing an asymptotic improvement to the bivariate Hensel Lifting algorithm described in Section 1.2. He created an algorithm which improved upon the bottleneck of Algorithm 2, namely Step 6

$$e_k \leftarrow A - \prod_{i=1}^n f_i. \tag{1.24}$$

which is equivalent to calculating

$$e_k = A - \prod_{i=1}^n f_i^{(k)}.$$

The reason (1.24) is so costly is that it fully computes the product of the factors, $\prod_{i=1}^n f_i^{(k)}$, during every iteration of Algorithm 2. This is inefficient as many multiplications are repeated. As a reminder to the reader, during every iteration of Hensel Lifting, the algorithm finds a polynomial $c_k \in \mathbb{F}_p[x]$ such that

$$c_k = \sum_{i=1}^n \sigma_i \frac{\prod_{j=1}^n f_{j,0}}{f_{i,0}}. \tag{1.25}$$

We derived this formula in Section 1.2. The algorithm then solves the Diophantine equation (1.25) to find σ_i for $1 \leq i \leq n$. Next, having found the lifting coefficients, the algorithm performs the lifting operation on the $f_i^{(k)}$ factors.

Bernardin found a way to calculate c_k that does not involve calculating the error e_k. Let $A \in \mathbb{F}_p[x, y]$ with $d_x = \deg(A, x) > 0$ and $d_y = \deg(A, y) > 0$. We assume $A(x, y)$ is monic in x. The non-monic case will not be discussed. Bernardin's method calculates c_k using $O(n\, d_x^2\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$ as opposed to the $O(n\, d_x^2\, d_y^3)$ arithmetic operations of Algorithm 2. We explain Bernardin's idea.

First, recall we are attempting to find factors of A such that $A = \prod_{i=1}^{n} f_i^{(k)}$ for some $k \geq 1$, where each f_i is a power series around $(y-\alpha)$. Bernardin writes A as a power series around $(y-\alpha)$, namely,

$$A = \sum_{i=0}^{d_y} a_i(x)(y-\alpha)^i$$

where $a_0, a_1, ..., a_{d_y} \in \mathbb{F}_p[x]$. We will cover the method we use to compute this power series of A in Section 2.4. In the previous algorithm, we derived the formula for c_k from the formula

$$c_k \equiv \frac{A - \prod_{i=1}^{n} f_i^{(k)}}{(y-\alpha)^k} \pmod{(y-\alpha)}.$$

This implies that $c_k = \text{coeff}(A - \prod_{i=1}^{n} f_i^{(k)}, (y-\alpha)^k)$. As we have shown, $a_k = \text{coeff}(A, (y-\alpha)^k)$. So, we will let the remaining terms be defined as $\Delta = \text{coeff}(\prod_{i=1}^{n} f_i^{(k)}, (y-\alpha)^k)$. This means that we can define c_k as $c_k = a_k - \Delta \in \mathbb{F}_p[x]$. We now need a way to find Δ efficiently.

We will begin by describing the concept of *convolution* which is an alternative way to multiply two polynomials that calculates each coefficient of the product exclusively. Suppose we have polynomials $a = \sum_{i=0}^{d_1} a_i x^i, b = \sum_{i=0}^{d_2} b_i x^i$, and wish to calculate $c = ab = \sum_{j=0}^{d_1+d_2} c_j x^j$. Using convolution, we calculate the coefficients c_j as

$$c_j = \sum_{i=\max(0,j-d_1)}^{\min(j,d_2)} a_{j-i} b_i \quad \text{for} \quad 0 \leq j \leq d_1 + d_2. \tag{1.26}$$

Example 1.14. *We will use convolution to calculate the coefficient of x^3 for the product of $c = a \times b$ where $a = 4x^2 + 3x + 5$ and $b = 6x^3 + 2x^2 + 8$.*

Using (1.26), we calculate the sum

$$\text{coeff}(c, x^3) = \sum_{i=\max(0,j-d_1)}^{\min(j,d_2)} a_{j-i} b_i = \sum_{i=1}^{3} a_{j-i} b_i = a_2 b_1 + a_1 b_2 + a_0 b_3 = 4 \cdot 0 + 3 \cdot 2 + 5 \cdot 6 = 36.$$

△

It is obvious that we can easily extend this idea to work for bivariate polynomials in $\mathbb{F}_p[x, y]$. We can perform convolution using univariate multiplication in $\mathbb{F}_p[x]$ in terms of the variable y.

Example 1.15. *We will use convolution to calculate the coefficient of y^2 for the product of $c = a \times b$ where $a = (3x+5)y^2 + (4x-2)y + 5x$ and $b = (x-3)y^2 + (-4x+1)y + (3x+2)$.*

Using (1.26), we calculate the sum

$$\begin{aligned}\operatorname{coeff}(c, y^2) &= \sum_{i=\max(0,j-d_1)}^{\min(j,d_2)} a_{j-i}b_i = \sum_{i=0}^{2} a_{j-i}b_i = a_2b_0 + a_1b_1 + a_0b_2 \\ &= (3x+5)\times(3x+2) + (4x-2)\times(-4x+1) + (5x)\times(x-3) = -2x^2 + 18x + 8.\end{aligned}$$

$\triangle$

Bernardin uses this method of polynomial multiplication to compute Δ for $n \geq 2$ factors using as few multiplications as possible. As a forewarning to the reader, the method for calculating Δ is the most difficult part of this thesis. We were only able to understand it by working through multiple examples.

The most straightforward method of calculating Δ is to calculate the product of $f_1^{(k)} \times \cdots \times f_n^{(k)}$ and take the coefficient for the $(y-\alpha)^k$ term, or symbolically

$$\Delta = \operatorname{coeff}\left(\prod_{i=1}^{n} f_i^{(k)}, (y-\alpha)^k\right).$$

However, this is equivalent to calculating the error in the previous section. Suppose we use convolution to calculate Δ between the factor $f_n^{(k)}$ and the product of $f_1^{(k)} ... f_{n-1}^{(k)}$. Then

$$\Delta = \sum_{j=0}^{k}\left[f_{n,j} \times \operatorname{coeff}\left(\prod_{i=1}^{n-1} f_i^{(k)}, (y-\alpha)^{k-j}\right)\right].$$

This is a valid way to calculate Δ, however, we would have to calculate $\operatorname{coeff}(\prod_{i=1}^{n-1} f_i^{(k)}, (y-\alpha)^m)$ for $0 \leq m \leq k$. So ,at this point there is still no asymptotic improvement over Algorithm 2. However, as we are in the kth iteration of the main Hensel lifting loop, we have calculated $\operatorname{coeff}(\prod_{i=1}^{n-1} f_i^{(k)}, (y-\alpha)^m)$ for $0 \leq m \leq k-1$ in previous iterations. So, if we store those polynomials in some matrix, G, then we only need to calculate the coefficient for $m = k$, and we can extract the rest from G to save on additions and multiplications.

Next, we must calculate $\operatorname{coeff}(\prod_{i=1}^{n-1} f_i^{(k)}, (y-\alpha)^k)$ for some k. Using convolution, between $f_{n-1}^{(k)}$ and $f_1^{(k)} ... f_{n-2}^{(k)}$, we get

$$\operatorname{coeff}\left(\prod_{i=1}^{n-1} f_i^{(k)}, (y-\alpha)^k\right) = \sum_{j=0}^{k}\left[f_{n-1,j} \times \operatorname{coeff}\left(\prod_{i=1}^{n-2} f_i^{(k)}, (y-\alpha)^{k-j}\right)\right].$$

Similarly, we need to calculate coeff$(\prod_{i=1}^{n-2} f_i^{(k)}, (y-\alpha)^m)$ for $0 \le m \le k$, but since we have stored the values for $0 \le m \le k-1$, we only need to calculate coeff$(\prod_{i=1}^{n-2} f_i^{(k)}, (y-\alpha)^k)$. So, in every kth iteration of the main Hensel Lifting loop, we must calculate the sequence of

$$\text{coeff}\left(\prod_{i=1}^{r} f_i^{(k)}, (y-\alpha)^k\right) \tag{1.27}$$

for r from 2 to n, where Δ is equal to (1.27) when $r = n$.

We will now define the matrix G which will store the intermediate sub-products as we calculate Δ. Let G be the $n \times (d_y + 1)$ matrix of polynomials in $\mathbb{F}_p[x]$ whose (i,k) entry of G contains the polynomial

$$G_{i,k} = \text{coeff}\left(\prod_{j=1}^{i} f_j^{(k+1)}, (y-\alpha)^k\right).$$

Notice that the polynomial stored in $G_{i,k}$ represents the coefficient for the $(k+1)$st approximation of the factors, not the kth approximation. This will be relevant later in this section.

We will refer to the algorithm that calculates Δ as *CoefficientExtraction*. CoefficientExtraction uses convolution and univariate multiplication in $\mathbb{F}_p[x]$ to calculate $\Delta = \text{coeff}(\prod_{j=1}^{n} f_j^{(k)}, (y-\alpha)^k)$ in the kth iteration of the Hensel Lifting algorithm. The algorithm calculates Δ by calculating (1.27) for r from 2 to n while using polynomials stored in G to minimize the number of univariate polynomial multiplications. We present Bernardin's CoefficientExtraction algorithm as Algorithm 3.

We will cover the algebraic complexity of Algorithm 3 after stating Bernardin's Hensel Lifting algorithm. We now encounter a new problem. When we calculate Δ, we only have the kth approximation of the factors and not the $(k+1)$st approximation. This means there are several terms, corresponding to the n terms on the right hand side of (1.25), that are absent when we performed the convolutions stored in G. We need to add these terms to G in order to calculate Δ in future iterations of the main Hensel Lifting loop. We could use the CoefficientExtraction algorithm again after lifting the factors to their $(k+1)$st approximation. Algorithm 3 would calculate the correct polynomials and store them in G for the kth iteration without increasing the algebraic complexity, but we can be more efficient. We introduce the *CoefficientUpdate* algorithm, which updates the polynomials in G using the minimal number of arithmetic operations to include the missing terms. We present Bernardin's CoefficientUpdate algorithm as Algorithm 4.

Example 1.16. *We will demonstrate one iteration of the CoefficientExtraction and CoefficientUpdate algorithms to calculate Δ for the $k = 3$ iteration of our algorithm and store the neces-*

Algorithm 3: Bernardin's Coefficient Extraction Algorithm

Input: prime p, $\alpha \in \mathbb{F}_p$, $k \in \mathbb{Z}^+$, $f_1, f_2, ..., f_n \in \mathbb{F}_p[x, y]$, G an $n \times (d_y + 1)$ matrix of elements in $\mathbb{F}_p[x]$.
Output: $\Delta = \text{coeff}(\prod_{i=1}^{n} f_i, (y-\alpha)^k) \in \mathbb{F}_p[x]$, G an $n \times (d_y + 1)$ matrix of elements in $\mathbb{F}_p[x]$.

if $n = 2$ **then**
  $MIN \leftarrow \max(0, k - \deg(f_2, y))$;
  $MAX \leftarrow \min(k, \deg(f_1, y))$;
  $\Delta \leftarrow \sum_{j=MIN}^{MAX} \text{coeff}(f_1, (y-\alpha)^j) \cdot \text{coeff}(f_2, (y-\alpha)^{k-j})$; .... $O(kd_x^2)$
else
  $d \leftarrow \deg(f_1, y)$;
  $G_{1,k} \leftarrow \text{coeff}(f_1, (y-\alpha)^k)$;
  for i from 2 to n **do**
    $\delta \leftarrow d$;
    $d \leftarrow d + \deg(f_i, y)$;
    if $k \leq d$ **then**
      $MIN \leftarrow \max(0, k - \delta)$;
      $MAX \leftarrow \min(k, \deg(f_i, y))$;
      $G_{i,k} \leftarrow \sum_{j=MIN}^{MAX} G_{i-1,k-j} \times \text{coeff}(f_i, (y-\alpha)^j)$; .... $O(kd_x^2)$
    end
  end
  $\Delta \leftarrow G_{n,k}$;
end
return Δ, G;

Algorithm 4: Bernardin's Coefficient Update Algorithm

Input: prime p, $\alpha \in \mathbb{F}_p$, $k \in \mathbb{Z}^+$, $f_1, f_2, ..., f_n \in \mathbb{F}_p[y]$, G an $n \times d_y + 1$ matrix of elements in $\mathbb{F}_p$.
Output: G an $n \times d_y$ matrix of elements in $\mathbb{F}_p$.

if $n > 2$ **then**
  $t \leftarrow \text{coeff}(f_1, (y-\alpha)^k)$;
  $G_{1,k} \leftarrow t$;
  for i from 2 to n **do**
    // $t = f_{i,0} \sum_{j=1}^{i-1} f_{j,k} \prod_{\substack{j=m=1 \\ m \neq j}}^{i-1} f_{m,0} + f_{i,k} \prod_{j=1}^{i-1} f_{j,0}$
    $t \leftarrow \text{coeff}(f_i, (y-\alpha)^0) \times t + \text{coeff}(f_i, (y-\alpha)^k) \times G_{i-1,0}$; .... $O(d_x^2)$
    $G_{i,k} \leftarrow G_{i,k} + t$; .... $O(d_x^2)$
  end
end
return G;

sary coefficients in the matrix H. Let $p = 11, \alpha = 3, n = 3$, and the field be $\mathbb{F}_p$. We will find $\text{coeff}(f_1 \times f_2 \times f_3, (y-3)^3)$ *where*

$$f_1^{(3)} = x^2 + (5x-1)(y-3)^2 + (-2x-5)(y-3) + (-x-2),$$

$$f_2^{(3)} = x^2 + (-2x-2)(y-3)^2 + (-4x+3)(y-3) + (-4x+3), \text{ and}$$

$$f_3^{(3)} = x^2 + (-3x+4)(y-3)^2 + (4x+5)(y-3) + (5x+4).$$

The $n \times (d_y + 1)$ matrix G was input as a 3×10 matrix, where the first 4 rows can be seen in Table 1.4. Note the table is currently transposed to fit on the page more comfortably.

d_y \ n	1	2	3
0	$x^2 - x - 2$	$x^4 - 5x^3 + 5x^2 + 5x + 5$	$x^6 - 5x^3 - x^3 - 5x^2 + x - 2$
1	$-2x - 5$	$5x^3 - x^2 - 3x + 1$	$-2x^5 - 2x^4 - 4x^3 + 5x^2 + 5x - 4$
2	$5x - 1$	$3x^3 - 2x^2 - 5x - 3$	$-3x^4 + 5x^3 - 4x^2 + 3x + 2$
3	-	-	-

Table 1.4: The first 4 rows of matrix G which contains the intermediate products of calculating $f_1 \times f_2 \times f_3$

Following Algorithm 3, we calculate the entries of the 3rd row of G, namely $G_{1,3}, G_{2,3}$, and $G_{3,3}$. First, we define $G_{1,3}$ =$\text{coeff}(f_1, (y-\alpha)^3) = 0$. *Next, we calculate $G_{2,3}$ and $G_{3,3}$ as*

$$\begin{aligned}
G_{2,3} &= \sum_{j=1}^{2} G_{1,3-j} \times \text{coeff}(f_2, (y-3)^j) \\
&= G_{1,2} \times \text{coeff}(f_2, (y-3)^1) + G_{1,1} \times \text{coeff}(f_2, (y-3)^2) \\
&= \text{coeff}(f_1, (y-3)^2) \times \text{coeff}(f_2, (y-3)^1) \\
&= (5x-1) \times (-4x+3) + (-2x-5) \times (-2x-2) \\
&= -5x^2 - 4
\end{aligned}$$

$$
\begin{aligned}
G_{3,3} &= \sum_{j=0}^{2} G_{2,3-j} \cdot \text{coeff}(f_3, (y-3)^j) \\
&= G_{2,3} \cdot \text{coeff}(f_3, (y-3)^0) + G_{2,2} \cdot \text{coeff}(f_3, (y-3)^1) + G_{2,1} \cdot \text{coeff}(f_3, (y-3)^2) \\
&= \text{coeff}(f_1 \times f_2, (y-3)^3) \times \text{coeff}(f_3, (y-3)^0) + \text{coeff}(f_1 \times f_2, (y-3)^2) \times \text{coeff}(f_3, (y-3)^1) \\
&\quad + \text{coeff}(f_1 \times f_2, (y-3)^1) \times \text{coeff}(f_3, (y-3)^2) \\
&= (-5x^2-4)(x^2+5x+4) + (3x^3-2x^2-5x-3)(4x+5) + (5x^3-x^2-3x+1)(-3x+4) \\
&= 3x^4+5x^3-5x^2+5x-5
\end{aligned}
$$

So, we have updated the 3rd row of G as

3	0	$-5x^2-4$	$3x^4+5x^3-5x^2+5x-5$

At this point of Bernardin's algorithm, we would solve the Diophantine equation, then lift the factors f_1, f_2 and f_3. In this example, we lift them to

$$f_1^{(4)} = x^2 + (-4x+1)(y-3)^3 + (5x-1)(y-3)^2 + (-2x-5)(y-3) + (-x-2),$$

$$f_2^{(4)} = x^2 + (3x-3)(y-3)^3 + (-2x-2)(y-3)^2 + (-4x+3)(y-3) + (-4x+3), \text{ and}$$

$$f_3^{(4)} = x^2 + (-2x-5)(y-3)^3 + (-3x+4)(y-3)^2 + (4x+5)(y-3) + (5x+4).$$

So, now we need to add the values of $f_{1,3}f_{2,0}f_{2,0} + f_{1,0}f_{2,3}f_{2,0} + f_{1,0}f_{2,0}f_{2,3}$ to G. We do this by applying the CoefficientUpdate algorithm. By executing Algorithm 4, we get

$$t = \text{coeff}(f_1, (y-3)^3) = -4x+1$$

$$G_{3,1} = t = -4x+1$$

$$
\begin{aligned}
&t = \text{coeff}(f_2, (y-3)^0) \cdot t + \text{coeff}(f_2, (y-3)^3) \cdot G_{1,0} \\
&= (x^2-4x+3)(-4x+1) + (3x-3)(x^2-x-2) = -x^3+3x-2
\end{aligned}
$$

$$G_{3,2} = G_{3,2} + t = (-5x^2-4) + (-x^3+3x-2) = -x^3-5x^2+3x+5$$

$$t = \text{coeff}(f_3, (y-3)^0) \cdot t + \text{coeff}(f_3, (y-3)^3) \cdot G_{2,0} = -3x^5+3x^3$$

$$G_{3,3} = G_{3,3} + t = (3x^4+5x^3-5x^2+5x-5) + (-3x^5+3x^3) = -3x^5+3x^4-3x^3-5x^2+5x-5$$

This leads to the updated G:

d_y \ n	1	2	3
0	x^2-x-2	$x^4-5x^3+5x^2+5x+5$	$x^6-5x^3-x^3-5x^2+x-2$
1	$-2x-5$	$5x^3-x^2-3x+1$	$-2x^5-2x^4-4x^3+5x^2+5x-4$
2	$5x-1$	$3x^3-2x^2-5x-3$	$-3x^4+5x^3-4x^2+3x+2$
3	$-4x+1$	$-x^3-5x^2+3x+5$	$-3x^5+3x^4-3x^3-5x^2+5x-5$

Table 1.5: The first 4 rows of matrix G which contains the sub-products of calculating $f_1 \times f_2 \times f_3$ after using CoefficientUpdate

Note the polynomial in $G_{3,3}$ *in Table 1.4 is equivalent to* $\text{coeff}(f_1^{(3)} \times f_2^{(3)} \times f_3^{(3)}, (y-3)^3)$, *while the polynomial* $G_{3,3}$ *in Table 1.5 is equivalent to* $\text{coeff}(f_1^{(4)} \times f_2^{(4)} \times f_3^{(4)}, (y-3)^3)$. *To give an idea of what* G *looks like at the end of the algorithm, we present the complete matrix* G *in the* $k = 9$*th and final iteration of the Hensel Lifting algorithm as Table 1.6. The entries denoted by "..." represent non-zero polynomials.*

d_y \ n	1	2	3
0	x^2-x-2	$x^4-5x^3+5x^2+5x+5$	$x^6-5x^3-x^3-5x^2+x-2$
1	$-2x-5$	$5x^3-x^2-3x+1$	$-2x^5-2x^4-4x^3+5x^2+5x-4$
2	$5x-1$	$3x^3-2x^2-5x-3$	$-3x^4+5x^3-4x^2+3x+2$
3	$-4x+1$	$-x^3-5x^2+3x+5$	$-3x^5+3x^4-3x^3-5x^2+5x-5$
4	0	...	...
5	0	...	...
6	0	...	...
7	0	0	...
8	0	0	...
9	0	0	...

Table 1.6: The 10 rows of matrix G which contains the sub-products of calculating $f_1 \times f_2 \times f_3$

$\triangle$

The differences between Bernardin's algorithm and the classical bivariate Hensel Lifting iteration come down to how c_k is calculated. The classical version calculates c_k by subtracting the product of $f_1^{(k)} f_2^{(k)} ... f_n^{(k)}$ from $A(x, y)$ and dividing by $(y - \alpha)^k$, while Bernardin chose to transform A into a power series around $(y-\alpha)$, then subtract the necessary terms, Δ, from the coefficients of A. We state Bernardin's algorithm as Algorithm 5. The order terms on the right count arithmetic operations in $\mathbb{F}_p$.

Algorithm 5: Bernardin's Bivariate Hensel Lift Algorithm

Input: prime p, $\alpha \in \mathbb{F}_p$, $A \in \mathbb{F}_p[x,y]$ and $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ satisfying $(i) A, f_{1,0}, f_{2,0}, ..., f_{n,0}$ are monic in x, $(ii) A(y=\alpha) = f_{1,0} f_{2,0} ... f_{n,0}$ and $(iii) \gcd(f_{i,0}, f_{j,0}) = 1$ for $i \neq j$.

Output: $f_1, f_2, ..., f_n \in \mathbb{F}_p[x,y]$ such that $A = f_1\, f_2 \ldots f_n$ or FAIL.

$d_x \leftarrow \deg(A,x)$; $d_y \leftarrow \deg(A,y)$;

for $i = 1$ to n **do** $f_i^{(1)} \leftarrow f_{i,0}$; $df_i \leftarrow 0$; **end**

Compute $a_0, a_1, ..., a_{d_y} \in \mathbb{F}_p[x]$ s.t. $A = \sum_{k=0}^{d_y} a_k (y-\alpha)^k$; .... $O(d_x\, d_y^2)$

$G_{n,0} \leftarrow \text{coeff}(f_n, (y-\alpha)^0)$;

for $i = n-1$ by -1 to 1 **do**

  $G_{i,0} \leftarrow G_{i+1,0} \cdot \text{coeff}(f_i, (y-\alpha)^0)$; .... $O(d_x^2)$

end

for $k = 1$ to d_y **while** $\sum_{i=1}^{n} df_i < d_y$ **do**

  $\Delta \leftarrow$ CoefficientExtraction$(p, \alpha, k, f_1, f_2, ..., f_n, G) \in \mathbb{F}_p[x]$; .... $O(k\, n\, d_x^2)$

  $c_k \leftarrow a_k - \Delta$;

  if $\sum_{i=1}^{n} df_i = \deg(A,y)$ and $c_k \neq 0$ **then Return** FAIL; **end**

  if $c_k \neq 0$ **then**

    Solve $\sum_{i=1}^{n} \sigma_i \frac{f_{1,0} f_{2,0} \cdots f_{n,0}}{f_{i,0}} = c_k$ in $\mathbb{F}_p[x]$ for $\sigma_1, \sigma_2, ..., \sigma_n \in \mathbb{F}_p[x]$; .... $O(n\, d_x^2)$

    for *i=1..n* **do**

      $f_{i,k} \leftarrow \sigma_i$;

      $f_i \leftarrow f_i + f_{i,k}(y-\alpha)^k$; .... $O(k\, d_x)$

      $df_i \leftarrow \deg(f_i, y)$;

    end

    CoefficentUpdate$(f_1, f_2, ..., f_n, (y-\alpha)^k, G)$; .... $O(n\, d_x^2)$

  end

end

if $\sum_{i=1}^{n} df_i = d_y$ **then Return** $f_1, f_2, ..., f_n$ **else return** FAIL; **end**

1.3.1 Cost of Bernardin's algorithm

We will consider the algebraic complexity of the *CoefficientExtraction* and *CoefficientUpdate* algorithms, Algorithms 3 and 4, before computing the cost of Bernardin's algorithm. Note, for the sake of historical context, we will perform the analysis suboptimally, that is, we can reduce the arithmetic operations of most of the sub-algorithms in Bernardin's algorithm, but the optimal analysis was not done in [2]. Therefore, we will not do optimal calculations here. We will perform optimal calculations during the analysis of our new algorithm in Section 3.2.

Consider the cost of performing the CoefficientExtraction algorithm. Suppose $d_x = \deg(A,x), d_y = \deg(A,y)$, and $\deg(f_i, x) < d_x$ for $1 \leq i \leq n$. The cost of the algorithm is clearly dominated by the number of multiplications done in Step 6 of CoefficientExtraction if $n = 2$ or Step 16 if $n > 2$. If $n = 2$, then at most $k+1$ univariate multiplications are performed. As the degree of each polynomial is bounded above by d_x, the algebraic complexity of Step 6 in Algorithm 3 is at most $(k+1)O(d_x^2) = O(kd_x^2)$. Similarly, if $n > 2$, at most $k+1$ univariate multiplications are performed. As the degree of each polynomial is bounded above by d_x, and each of the $k+1$ multiplications is done $n-1$ times, the algebraic cost of Step 17 is $(k+1)(n-1)O(d_x^2) = O(n\,k\,d_x^2)$. Therefore, the cost of the CoefficientExtraction algorithm is $O(n\,k\,d_x^2)$.

Consider the cost of performing the CoefficientUpdate algorithm. The cost of the algorithm is defined by two univariate multiplications in Step 8 of Algorithm 4. As both polynomials have a degree of at most d_x, this step performs $O(d_x^2)$ multiplications and $O(d_x)$ additions. The multiplications are done $n-1$ times, so the cost of the CoefficientUpdate algorithm is $nO(d_x^2) = O(nd_x^2)$.

The cost of Bernardin's algorithm, Algorithm 5, is dominated by the cost of performing CoefficientExtraction in Step 11. As Algorithm 5 must calculate Δ for each of the d_y coefficients of A to ensure we have found the factorization of A. Therefore, the main loop is executed d_y times and the algebraic complexity of Bernardin's algorithm is $\sum_{k=1}^{d_y} O(n\,k\,d_x^2) = O(n\,d_x^2\,d_y^2)$.

In theory, the CoefficientExtraction and CoefficientUpdate algorithms calculate the product of $f_1 f_2 ... f_n$ exactly once. As we are unaware if a proof of this concept exists, we leave it as a conjecture.

Conjecture 1.17. *In Bernardin's algorithm, the multiplications computed by the initialization of G, the CoefficientExtraction algorithm and the CoefficientUpdate algorithm are the same as the multiplications in $\mathbb{F}_p[x]$ needed to calculate the product $f_1 \times f_2 \times \cdots \times f_n$ in $\mathbb{F}_p[x, y]$. If $A = \prod f_i$, $d_x = \deg(A, x)$ and $d_y = \deg(A, y)$ then the cost of these multiplications is bounded by $O(d_x^2\,d_y^2)$ arithmetic operations in $\mathbb{F}_p$ using classical multiplication in $\mathbb{F}_p[x]$.*

Chapter 2

Tools

In this chapter, we will describe several algorithms that we will be using in our proposed algorithm. We will discuss the method and calculate the arithmetic operations done by each algorithm. The four well-known algorithms we will cover are:

- Horner's method for evaluating polynomials
- Solving polynomial Diophantine equations
- Polynomial interpolation
- Power series base conversion

2.1 Horner's method

Horner's method refers to a polynomial evaluation method named after William George Horner. This algorithm is much older than Horner; he himself ascribed it to Joseph-Louis Lagrange, but it can be traced back many hundreds of years to Chinese and Persian mathematicians [1].

Consider a polynomial $p(x) \in \mathbb{Z}[x]$ given by

$$p(x) = \sum_{i=0}^{d} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d. \tag{2.1}$$

To perform polynomial evaluation at $x = \alpha$, for some $\alpha \in \mathbb{Z}$, we could first calculate $\alpha^2, \alpha^3, ..., \alpha^d$ using $d-1$ multiplications. We then perform an additional d scalar multiplications followed by d additions to sum up the $d+1$ terms. Performing evaluations this way results in $2d-1$ multiplications and d additions. Consider writing (2.1) as

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_d)...)). \tag{2.2}$$

If we perform the same polynomial evaluation on (2.2), we require exactly d multiplications and d additions. Horner's method performs $O(d)$ arithmetic operations in $\mathbb{Z}$ when applied to univariate polynomials over a coefficient ring.

2.2 Solving polynomial Diophantine equations

It is vital that we have an algorithm which can solve polynomial Diophantine equations quickly. If we are given $f_{10}, f_{20}, ..., f_{n0}, c \in \mathbb{F}_p[x]$ for some prime p such that $\gcd(f_{i0}, f_{j0}) = 1$ in $\mathbb{F}_p[x]$ for $i \neq j$, then we need to be able to solve a Diophantine equation of the form

$$\sum_{i=1}^{n} \sigma_i \frac{\prod_{j=1}^{n} f_{j,0}}{f_{i,0}} = c \tag{2.3}$$

for $\sigma_1, ..., \sigma_n \in \mathbb{F}_p[x]$. Hensel's Lemma assumes that a solution to (2.3) exists when $n = 2$, and every iteration of Hensel Lifting stated in this thesis needs to be able to solve a Diophantine equation for $n \geq 2$ factors. In Chapter 1, when performing Historical Hensel Lifting for two factors, we needed to solve (2.3) for the $n = 2$ factor case. For Bernardin's algorithm and our algorithm, which have n factors, we need an algorithm which can solve the $n \geq 2$ case. We will start by defining the algorithm for the $n = 2$ factor case and then extend it to work for $n > 2$ factors.

2.2.1 The $n = 2$ factor case

Suppose we are given a prime p and polynomials $u_0, w_0, c \in \mathbb{F}_p[x]$ such that $\gcd(u_0, w_0) = 1$ in $\mathbb{F}_p[x]$ and $\deg(c, x) < \deg(u_0, x) + \deg(w_0, x)$. We need to solve the Diophantine equation

$$\sigma w_0 + \tau u_0 = c \tag{2.4}$$

for some unique polynomials $\sigma, \tau \in \mathbb{F}_p[x]$. We will remain consistent with the notation used in Chapter 1 by defining $f_{1,0} = u_0$ and $f_{2,0} = w_0$.

This is the Diophantine equation that appears in the two factor case of Hensel Lifting in Section 1.1. Before we describe how to find a solution to (2.4), we restate Theorem 1.3 from Section 1.1.1 for a field F, and give a formal proof. The theorem and proof are presented as Theorem 2.6 in [7].

Theorem 1.3. *Let $F[x]$ be the Euclidean domain of univariate polynomials over a field F. Let $a(x), b(x) \in F[x]$ be given nonzero polynomials and let $g(x) = \gcd(a, b) \in F[x]$. Then for any given polynomial $c(x) \in F[x]$ such that $g|c$ there exist unique polynomials $\sigma(x), \tau(x) \in F[x]$ such that*

$$\sigma a + \tau b = c \text{ and} \tag{1.2}$$

$$\deg(\sigma, x) < \deg(b, x) - \deg(g, x). \tag{1.3}$$

Moreover, if $\deg(c, x) < \deg(a, x) + \deg(b, x) - \deg(g, x)$ *then* τ *satisfies*

$$\deg(\tau, x) < \deg(a, x) - \deg(g, x). \tag{1.4}$$

Proof. (Existence): The Extended Euclidean Algorithm can be applied to compute polynomials $s(x), t(x) \in F[x]$ satisfying the equation

$$sa + tb = g.$$

Then since $g|c$, it is easily seen that

$$(sc/g)a + (tc/g)b = c. \tag{2.5}$$

We therefore have a solution of equation (1.2), say $\hat{\sigma} = sc/g$ and $\hat{\tau} = tc/g$. However, the degree constraint of (1.3) will not in general be satisfied by this solution, so we will proceed to show how to reduce the degree. Writing (2.5) in the form

$$\hat{\sigma}(a/g) + \hat{\tau}(b/g) = c/g \tag{2.6}$$

we apply Euclidean division of $\hat{\sigma}$ by (b/g) yielding $q, r \in F[x]$ such that

$$\hat{\sigma} = (b/g)q + r \tag{2.7}$$

where $r = 0$ or $\deg(r, x) < \deg(b, x) - \deg(g, x)$. Now define $\sigma = r$ and note that (1.3) is satisfied. Also define $\tau = \hat{\tau} + q(a/g)$. It is easily verified by using (2.6) and (2.7) that

$$\sigma(a/g) + \tau(b/g) = c/g.$$

Equation (1.2) follows immediately.

(Uniqueness): Let $\sigma_1(x), \tau_1(x) \in F[x]$ and $\sigma_2(x), \tau_2(x) \in F[x]$ be two pairs of polynomials satisfying (1.2) and (1.3). The two different solutions of (1.2) can be written in the form

$$\sigma_1(a/g) + \tau_1(b/g) = c/g \text{ and } \sigma_2(a/g) + \tau_2(b/g) = c/g$$

which yields on subtraction

$$(\sigma_1 - \sigma_2)(a/g) = -(\tau_1 - \tau_2)(b/g). \tag{2.8}$$

Now, since a/g and b/g are relatively prime it follows from equation (2.8) that

$$(b/g) | (\sigma_1 - \sigma_2). \tag{2.9}$$

But, from the degree constraint (1.3) satisfied by σ_1 and σ_2 it follows that

$$\deg(\sigma_1 - \sigma_2) < \deg(b/g, x). \tag{2.10}$$

Now (2.9) and (2.10) together imply that $\sigma_1 - \sigma_2 = 0$. It then follows from (2.8) that $\tau_1 - \tau_2 = 0$ since $b/g \neq 0$. Therefore $\sigma_1 = \sigma_2$ and $\tau_1 = \tau_2$.

(Final Degree Constraint:) It remains to prove (1.4). From (1.2) we can write

$$\tau = (c - \sigma a)/b$$

so that

$$\deg(\tau, x) = \deg(c - \sigma a, x) - \deg(b, x). \tag{2.11}$$

Note, if $\deg(c, x) \geq \deg(\sigma a)$ then from (2.11)

$$\deg(\tau, x) \leq \deg(c, x) - \deg(b, x) < \deg(a, x) - \deg(g, x)$$

as long as $\deg(c, x) < \deg(a, x) + \deg(b, x) - \deg(g, x)$ as stated. Otherwise, if $\deg(c, x) < \deg(\sigma a, x)$ (in which case the stated degree bound for c also holds because of (1.3)) then from (2.11)

$$\begin{aligned}
\deg(\tau, x) &= \deg(\sigma a, x) - \deg(b, x) \\
&= \deg(\sigma, x) + \deg(a, x) - \deg(b, x) \\
&< \deg(a, x) - \deg(g, x) \ \text{ since } \ \deg(\sigma, x) < \deg(b, x) - \deg(g, x)
\end{aligned}$$

where the last inequality follows from (1.3). Thus (1.4) is proved. □

Now that we have proved Theorem 1.3, we know a unique solution to (2.4) exists. We can now describe the necessary algorithm. Let $h = \gcd(u_0, w_0)$. We are given two polynomials $u_0, w_0 \in \mathbb{F}_p[x]$. We can apply the Extended Euclidean Algorithm (EEA) to solve for the greatest common divisor of u_0 and w_0. As a consequence of the algorithm, we find two polynomials $s, t \in \mathbb{F}_p[x]$ such that

$$sw_0 + tu_0 = h. \tag{2.12}$$

We assume that $\gcd(u_0, w_0) = 1$ in $\mathbb{F}_p[x]$ for Hensel Lifting. Now, we need the right hand side of (2.12) to be equal to c, so we can multiply both sides by c to get

$$csw_0 + ctu_0 = c. \tag{2.13}$$

We now have a solution to (2.4), $\sigma = cs$ and $\tau = ct$, however, this may not satisfy the condition that $\deg(\sigma, x) < \deg(u_0, x)$. To ensure this condition holds, we use the polynomial division $cs \div u_0$ to obtain the quotient q and the remainder r where $cs = qu_0 + r$ and $r = 0$ or $\deg(r, x) < \deg(u_0, x)$. We can then substitute $cs = qu_0 + r$ into (2.13) and rearrange to get

$$\begin{aligned} &(qu_0 + r)w_0 + ctu_0 = c \\ \Rightarrow\ &\underbrace{r}_{\sigma} w_0 + (\underbrace{qw_0 + ct}_{\tau})u_0 = c. \end{aligned} \tag{2.14}$$

We have now found a solution to the Diophantine equation, namely $\sigma = r$ and $\tau = qw_0 + ct$. In addition, $\deg(\tau, x) < \deg(w_0, x)$ by Theorem 1.3. One last problem arises from this definition of τ. Calculating τ using $\tau = qw_0 + ct$ can be very expensive as the degrees of q and t can be large. Notice that we can rearrange (2.14) to get an alternative way to calculate τ as $\tau = \frac{c-\sigma w_0}{u_0}$.

We claim that calculating $\tau = \frac{c-\sigma w_0}{u_0}$ uses fewer arithmetic operations than calculating $\tau = qw_0 + ct$. Suppose $d_1 = \deg(u_0, x), d_2 = \deg(w_0, x)$, and $d_3 = \deg(c, x)$ such that $d_1 \leq d_2$. We will assume $d_3 < d_1 + d_2$ so that $\deg(\tau, x) < \deg(w_0, x)$ by Theorem 1.3. Before we can do proper analysis, we need to find the degrees of polynomials s, t, cs, q, r, σ, and $c - \sigma w_0$. According to Section 3.3 of [6], using the EEA we can find $s, t \in \mathbb{F}_p[x]$ such that $\deg(s, x) < \deg(u_0, x) = d_1$ and $\deg(t, x) < \deg(w_0, x) = d_2$. Then $\deg(cs, x) = \deg(c, x) + \deg(s, x) < d_3 + d_1$. Using the division algorithm, we find quotient q and remainder r such that $\deg(q, x) = \deg(cs, x) - \deg(u_0) < d_3 + d_1 - d_1 = d_3$ and $\deg(r, x) < \deg(u_0) = d_1$. Then, $\deg(\sigma, x) = \deg(r, x) < d_1$. Finally, we consider $\deg(c-\sigma w_0, x)$. As $\deg(c, x) = d_3 < d_1+d_2$ and $\deg(\sigma w_0, x) = \deg(\sigma, x)+\deg(w_0, x) < d_1+d_2$, we have $\deg(c - \sigma w_0, x) < d_1 + d_2$. To summarize

• $\deg(u_0, x) = d_1$	• $\deg(w_0, x) = d_2$	• $\deg(c, x) = d_3$
• $\deg(s, x) < d_1$	• $\deg(t, x) < d_2$	• $\deg(cs, x) < d_3 + d_1$
• $\deg(q, x) < d_3$	• $\deg(r, x) < d_1$	• $\deg(\sigma, x) < d_1$
• $\deg(c - \sigma w_0, x) < d_1 + d_2$		

Consider the number of multiplications needed to find τ using $\tau = qw_0 + ct$ and $\tau = \frac{c-\sigma w_0}{u_0}$. We will be using classical polynomial multiplication for calculations. When calculating $\tau = qw_0 + ct$, we perform two multiplications where qw_0 requires at most $(d_3)(d_2+1)$ multiplications and where ct requires at most $(d_3+1)(d_2)$ multiplications. On the other hand, when calculating $\frac{c-\sigma w_0}{u_0}$ we need to calculate the multiplication σw_0 and the division $(c-\sigma w_0) \div u_0$. The multiplication σw_0 performs at most $d_1(d_2+1)$ multiplications and the division does at most $(d_1+1)(d_1+d_2-d_1+1) = (d_1+1)(d_2+1)$ multiplications. To summarize,

- qw_0 does $d_3(d_2+1) < (d_1+d_2)(d_2+1) \in O(d_2^2)$ multiplications
- ct does $(d_3+1)(d_2) < (d_1+d_2+1)(d_2+1) \in O(d_2^2)$ multiplications
- σw_0 does $d_1(d_2+1) \in O(d_1 d_2)$ multiplications
- $c - \sigma w_0 \div u_0$ does $(d_1+1)(d_2+1) \in O(d_1 d_2)$ multiplications

Then computing $\tau = qw_0 + ct$ performs $O(d_2^2)$ multiplications and $\tau = \frac{c-\sigma w_0}{u_0}$ does $O(d_1 d_2)$ multiplications. Since we are assuming that $d_1 \leq d_2$, it is clear that calculating $\tau = \frac{c-\sigma w_0}{u_0}$ performs fewer arithmetic operations than calculating $\tau = qw_0 + ct$. The latter is much better if $d_1 \ll d_2$.

We present the algorithm to solve the polynomial Diophantine equation for two factors as Algorithm 6. The number of arithmetic operations done by each step is stated on the right.

Algorithm 6: Polynomial Diophantine equation algorithm: 2 factor case

1 **Input**: a prime p, $u_0, w_0 \in \mathbb{F}_p[x]$ such that $\gcd(u_0, w_0) = 1$ in $\mathbb{F}_p[x]$,$c \in \mathbb{F}_p[x]$ where $d_1 = \deg(u_0, x), d_2 = \deg(w_0, x)$, and $\deg(c, x) < d_1 + d_2$.

2 **Output**: $\sigma, \tau \in \mathbb{F}_p[x]$.

3 Solve $sw_0 + tu_0 = 1$ for $s, t \in \mathbb{F}_p[x]$ using the EEA; . . . . $O(d_1 d_2)$

4 Compute cs; . . . . $O(d_1 d_2)$

5 Solve $cs = qu_0 + r$ for $q, r \in \mathbb{F}_p[x]$ using the Division Algorithm; . . . . $O(d_1 d_2)$

6 $\sigma \leftarrow r$;

7 $\tau \leftarrow \frac{c-\sigma w_0}{u_0}$; . . . . $O(d_1 d_2)$

8 **return** σ, τ

We consider the total arithmetic operations in $\mathbb{F}_p$ needed to perform Algorithm 6. Let $d_1 = \deg(u_0, x), d_2 = \deg(w_0, x)$ and $d_3 = \deg(c, x)$ such that $d_1 \leq d_2$ and $d_3 < d_1 + d_2$. The first subroutine that is used is the Extended Euclidean Algorithm in Step 3. The EEA has a known cost of $O(nm)$ where m and n are the degrees of the two input polynomials to the EEA: see Theorem 3.16 of [6]. Therefore, the EEA does $O(d_1 d_2)$ arithmetic operations in $\mathbb{F}_p$. In addition, [6] states that the EEA calculates some $s, t \in \mathbb{F}_p[x]$ that satisfy $sw_0 + tu_0$ such that $\deg(s, x) < \deg(u_0, x) = d_1$ and $\deg(t, x) < \deg(w_0, x) = d_2$ using $O(d_1 d_2)$ arithmetic operations in $\mathbb{F}_p$.

Consider the cost of calculating cs. We know that $\deg(c, x) = d_3$ and $\deg(s, x) < d_1$. So, using classical multiplication, calculating cs does $(d_3+1)(d_1) < (d_1+d_2+1)(d_1) \in O(d_1 d_2)$ multiplications.

Next, we consider the number of multiplications done by the division algorithm in Step 5. The number of multiplications performed by the division algorithm on univariate polynomials in $\mathbb{F}_p[x]$ is $(m+1)(n-m+1)$ where n is the degree of the dividend and m is the degree of the divisor. See [6]. The dividend, cs, has $\deg(cs, x) < d_3 + d_1$ and the divisor, u_0, has $\deg(u_0, x) = d_1$. Therefore, in the worst case, the number of multiplications done is $(d_1+1)(d_3+d_1-d_1+1) = (d_1+1)(d_3+1) < (d_1+1)(d_1+d_2+1)$. As $d_1 \leq d_2$, performing polynomial division in Step 5 uses $O(d_1 d_2)$ multiplications.

Finally, we consider the calculation of τ. We have already shown above that computing τ using $\frac{c-\sigma w_0}{u_0}$ does $O(d_1 d_2)$ multiplications in $\mathbb{F}_p$.

We can find the total cost of Algorithm 6 by adding up the costs of the individual steps. As each step has a algebraic complexity of $O(d_1 d_2)$, the algebraic complexity of Algorithm 6 is $O(d_1 d_2)$.

2.2.2 The $n > 2$ factor case

We will now cover the case of solving a polynomial Diophantine equation for $n > 2$ factors which is needed to perform Hensel Lifting for $n > 2$ factors.

Suppose we are given a prime p, and polynomials $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ such that $\gcd(f_{i,0}, f_{j,0}) = 1$ in $\mathbb{F}_p[x]$ for all $i \neq j$. Let $U(x) = \prod_{j=1}^{n} f_{j,0}$ and $U_i(x) = \frac{U(x)}{f_{i,0}}$ for $1 \leq i \leq n$. Given some polynomial $c(x) \in \mathbb{F}_p[x]$, we need to solve the equation

$$\sigma_1 U_1 + \sigma_2 U_2 + \sigma_3 U_3 + ... + \sigma_n U_n = c \tag{2.15}$$

for $\sigma_1, \sigma_2, ..., \sigma_n \in \mathbb{F}_p[x]$ where $\deg(\sigma_i, x) < \deg(f_{i,0}, x)$ for $1 \leq i \leq n$.

To solve (2.15), we will take advantage of the method used to solve Diophantine equations for the two factor case in the previous section. That method can solve an equation of the form $\sigma a + \tau b = c$ for unique $\sigma, \tau \in \mathbb{F}_p[x]$ given $\gcd(a, b) = 1$ and $\deg(c, x) < \deg(a, x) + \deg(b, x)$. If we factor polynomial $f_{1,0}$ from the rightmost $n-1$ elements of (2.15), we get

$$\sigma_1 U_1 + f_{1,0} \underbrace{\left(\sigma_2 \frac{U_2}{f_{1,0}} + \sigma_3 \frac{U_3}{f_{1,0}} + ...\sigma_n \frac{U_n}{f_{1,0}}\right)}_{\tau_1} = c. \tag{2.16}$$

To solve (2.16), since $\gcd(U_1, f_{1,0}) = 1$, we solve $\sigma_1 U_1 + \tau_1 f_{1,0} = c$ using the two factor method. We obtain σ_1 and τ_1 with $\deg(\sigma_1, x) < \deg(f_{1,0}, x)$ as required.

Next, we notice from (2.16) that

$$\sigma_2 \frac{U_2}{f_{1,0}} + \sigma_3 \frac{U_3}{f_{1,0}} + \cdots + \sigma_n \frac{U_n}{f_{1,0}} = \tau_1. \tag{2.17}$$

This is a polynomial Diophantine equation in one less term where τ_1 is known. If we factor polynomial $f_{2,0}$ from the rightmost $n-2$ elements, we get

$$\sigma_2 \frac{U_2}{f_{1,0}} + f_{2,0} \underbrace{\left(\sigma_3 \frac{U_3}{f_{1,0} f_{2,0}} + \cdots + \sigma_n \frac{U_n}{f_{1,0} f_{2,0}}\right)}_{\tau_2} = \tau_1. \tag{2.18}$$

Now we solve $\sigma_2 \frac{U_2}{f_{1,0}} + f_{2,0}\tau_2 = \tau_1$ for σ_2 and τ_2 using the two factor method. Iterating this process, we can solve for all polynomials $\sigma_1, \sigma_2, ..., \sigma_n$. We present the algorithm which can solve a Diophantine equation for n factors as Algorithm 7.

Algorithm 7: Multi-Diophantine Polynomial equation Algorithm

Input: prime p, $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ satisfying $\gcd(f_{i,0}, f_{j,0}) = 1$ in $\mathbb{F}_p[x]$ for $i \neq j$, $c \in \mathbb{F}_p[x]$.

Output: $\sigma_1, \sigma_2, ..., \sigma_n \in \mathbb{F}_p[x]$.

$M_n \leftarrow 1$;

for $i = n-1$ by -1 to 1 **do** $M_i \leftarrow M_{i+1} \times f_{i+1,0}$ **end**; .... $O(d_x^2)$

$c_1 \leftarrow c$;

//Solve $\sigma_i M_i + \tau_i f_{i,0} = c_i$

for $i = 1$ to $n-1$ **do**

  Solve $s_i M_i + t_i f_{i,0} = 1$ for $s_i, t_i \in \mathbb{F}_p[x]$ using the EEA; .... $O(d_x^2)$

  $\sigma_i \leftarrow (c_i \cdot s_i)$ rem $f_{i,0}$; .... $O(d_x^2)$

  $\tau_i \leftarrow (c_i - \sigma_i M_i)$ quo $f_{i,0}$; .... $O(d_x^2)$

  $c_{i+1} \leftarrow \tau_i$;

end

$\sigma_n \leftarrow c_n$;

return $\sigma_1, \sigma_2, ..., \sigma_n$;

Now that we have defined the algorithm for solving a univariate, polynomial Diophantine equation for n factors, we must calculate the number of arithmetic operations it uses. Bernardin found the cost of Algorithm 7 to be $O(n\, d_x^2)$, because every step in the loop has a cost of at most $O(d_x^2)$ and the loop is performed $n-1$ times. In [2], Bernardin states that solving the Diophantine equation (2.15) "has a cost of $O(m)$ multiplications in the coefficient ring $\mathbb{F}_p[x]$ and thus a total cost of $O(m\, M(n))$ operations in $\mathbb{F}_p$, where $M(n)$ is the complexity of multiplying two univariate polynomials of degree n". Bernardin uses m to represent the number of factors where we use n and n to represent the degree of the univariate polynomials in $\mathbb{F}_p[x]$ where we use d_x. However, we have found that by using $\sum_{i=1}^{n} \deg(f_i, x) = d_x$, and by calculating τ_i using $\tau_i = \frac{c_i - \sigma_i M_i}{f_{i,0}}$ instead of using $\tau_i = qM_0 - c_i t_i$, we can reduce the cost of the algorithm to $O(d_x^2)$. As Bernardin's analysis was suboptimal, we will not recite his analysis of the multi-Diophantine equation. We will perform the complexity analysis of Algorithm 7 in Section 3.2.7.

2.3 Polynomial interpolation

The polynomial interpolation problem is, given n data points $(x_1, y_1), ..., (x_n, y_n)$, to find a polynomial $f(x)$ that interpolates the data, that is $f(x_i) = y_i$ for $1 \le i \le n$. The following well-known theorem gives us existence and uniqueness conditions for $f(x)$.

Theorem 2.1. *Let x_i, y_i, $1 \le i \le n$ be elements of a field F where the x_i are distinct. Then there exists a unique polynomial $f(x) \in F[x]$ of degree at most $n-1$ such that $y_i = f(x_i), 1 \le i \le n$.*

Proof. See Theorem 2.5.7 of [19]. □

There are many ways to compute the polynomial f in Theorem 2.1. The two most well-known methods of interpolation are Newton interpolation and Lagrange interpolation. Both methods perform $O(n^2)$ arithmetic operations in F. We will be using the Lagrange interpolation method in $\mathbb{Z}_p$. While Lagrange interpolation is named after Joseph Louis Lagrange, who published it in 1795, the method was first discovered in 1779 by Edward Waring [24]. It is also an easy consequence of a formula published in 1783 by Leonhard Euler [13].

Consider a polynomial $f = \sum_{i=0}^{d-1} a_i x^i$ where $a_i \in \mathbb{F}_p$. The Lagrange basis polynomials, denoted L_i, are polynomials in $\mathbb{Z}_p[x]$ with degree less than d having the property that $L_i(x_j) = 0$ if $i \neq j$ and $L_i(x_j) = 1$ when $i = j$. So

$$f = \sum_{i=1}^{d} y_i L_i \tag{2.19}$$

is a polynomial of degree less than d such that $f(x_i) = y_i$ for all i. The interpolating polynomial with this degree constraint is unique, since the difference of two such polynomials has degree less than d and d roots, hence is the zero polynomial.

Let (x_i, y_i) be the data in Theorem 2.1. Let

$$M(x) = \prod_{i=1}^{d}(x - x_i)$$

and let

$$M_i(x) = \frac{M(x)}{(x - x_i)} \text{ for } 1 \leq i \leq d.$$

The M_i are called the Lagrange basis polynomials. By construction, they have the property that $M_i(x_j) = 0$ if $i \neq j$. Let $\alpha_i = M_i(x_j)$ for $i = j$. We need them to have the property that $M_i(x_i) = 1$ for $1 \leq i \leq d$. So, we can let $L_i = M_i/\alpha_i$ to get the desired result. Since we are working in a field, multiplicative inverses exist, so $L_i \in \mathbb{Z}_p[x]$. Therefore, we have found the polynomials in $\mathbb{Z}_p[x]$ such that $L_i(x_j) = 0$ if $i \neq j$ and $L_i(x_j) = 1$ if $i = j$.

Now, let

$$f(x) = \sum_{i=1}^{d} y_i L_i.$$

So, $f(x)$ is a polynomial of degree at most $d-1$ such that $f(x_i) = 0 + 0 + ... + (1 \times y_i) + ... + 0 = y_i$. Therefore, we have found the desired basis polynomials. We now consider the cost of Lagrange interpolation. We present Theorem 2.2 as a formal statement of the cost of the Lagrange interpolation method.

Theorem 2.2. *Evaluating a polynomial $f \in F[x]$ of degree less than d at d distinct points $x_1, ..., x_d \in F$ or computing an interpolating polynomial at these points can be performed with $O(d^2)$ operations in F. More precisely, evaluation takes $2d^2 - 2d$ operations, and Lagrange interpolation uses $7d^2 - 8d + 1$ operations.*

Proof. See Theorem 5.1 in [6]. □

2.4 Base conversion

In this section, we are going to discuss performing a base conversion on a finite power series of the form

$$\sum_{i=0}^{d} b_i y^i = a_0 + a_1(y - \alpha) + a_2(y - \alpha)^2 + \cdots + a_d(y - \alpha)^d \tag{2.20}$$

where a_n represents the coefficient of the nth term and α is a constant. a_n is independent of y and may be expressed as a function of n. The power series we are interested in is the power series for some $A \in \mathbb{F}_p[x, y]$ about some $\alpha \in \mathbb{F}_p$ for some prime p. If we let $d_y = \deg(A, y)$, then A can be represented as

$$A(x, y) = \sum_{i=0}^{d_y} a_i (y - \alpha)^i \tag{2.21}$$

where $a_0, a_1, ..., a_{d_y} \in \mathbb{F}_p[x]$. There are several algorithms that we can use to find $a_0, a_1, ..., a_{d_y}$. The most straightforward method is to repeatedly perform polynomial division with remainder. If we divide A by $(y - \alpha)$ to get $A = q(y - \alpha) + r$, we obtain $r = a_0$ and $q = \sum_{i=0}^{d_y - 1} a_{i+1}(y - \alpha)^i$. By dividing q by $(y - \alpha)$, we get $q = q_1(y - \alpha) + r_1$ which gives us $a_1 = r_1$. Repeatedly dividing by $(y - \alpha)$ leads to an algorithm that does $O(d_x d_y^2)$ multiplications in $\mathbb{F}_p$. Shaw and Traub reduce this to $O(d_x d_y)$ multiplications and $O(d_x d_y^2)$ additions. As multiplications are more costly, we will use Shaw and Traub's method as described in [11].

Consider the Taylor representation of $A(x, y)$. By Taylor's theorem, the desired coefficients are given by the derivatives of A at $y = \alpha$, namely

$$A(x, y - \alpha) = A(x, \alpha) + \frac{\partial A}{\partial y}(x, \alpha)(y - \alpha) + \frac{\partial^2 A}{\partial y^2}(x, \alpha)(y - \alpha)^2/2! + \cdots + \frac{\partial^n A}{\partial y^n}(x, \alpha)(y - \alpha)^n/n!.$$

So, the problem is equivalent to evaluating A and all its derivatives at $y = \alpha$. Note that this method needs $p > n$.

If we write $A(x, y) = q(x, y)(y + \alpha) + r(x)$, then $A(x, y - \alpha) = q(x, y - \alpha)y + r(x)$; so $r(x)$ is the univariate coefficient of $A(x, y - \alpha)$, and the problem reduces to finding the coefficients of $q(x, y - \alpha)$, where $q(x, y)$ is a known polynomial of degree $d_y - 1$. Thus the following algorithm is indicated:

Algorithm 8: Shaw and Traub's nonoptimal bivariate base conversion algorithm

Input: prime p, $\alpha \in \mathbb{F}_p$, $d_y \in \mathbb{Z}^+$, $a_0, a_1, ..., a_{d_y} \in \mathbb{F}_p[x]$.
Output: $v_0, v_1, ..., v_{d_y} \in \mathbb{F}_p[x]$.

$v_j \leftarrow a_j$ for $0 \le j \le d_y$; .... $O(d_x\, d_y)$
for $k = 0$ to $d_y - 1$ **do**
    for $j = d_y - 1$ to k **do**
        $v_j \leftarrow v_j + \alpha v_{j+1}$; .... $O(d_x)$
    end
end
return $v_0, ..., v_{d_y}$;

At the conclusion of the algorithm, we have $A(x, y-\alpha) = v_0 + v_1(y-\alpha) + ... + v_{d_y}(y-\alpha)^{d_y}$. This procedure was a principal part of Horner's root-finding method, and when $k = 0$ it is exactly Horner's method for evaluating $A(x, \alpha)$.

If $d_x = \deg(A, x)$ and $d_y = \deg(A, y)$, we perform Step 6 $d_y + (d_y - 1) + ... + 1 = \frac{d_y(d_y+1)}{2}$ times. In each call, we perform at most $d_x + 1$ multiplication and $d_x + 1$ additions. Therefore this algorithm does $(d_x + 1)(d_y^2 + d_y)/2$ multiplications and $(d_x + 1)(d_y^2 + d_y)/2$ additions. Notice that if $\alpha = 1$ we may avoid all the multiplications. Fortunately, we can reduce the general problem to the case $\alpha = 1$ by introducing comparatively few multiplications and divisions. We present the modified version of Algorithm 8 as Algorithm 9.

Algorithm 9: Shaw and Traub's bivariate optimal base conversion algorithm

Input: prime p, a nonzero $\alpha \in \mathbb{F}_p$, $d_y \in \mathbb{Z}^+$, $a_0, a_1, ..., a_{d_y} \in \mathbb{F}_p[x]$.
Output: $v_0, v_1, ..., v_{d_y} \in \mathbb{F}_p[x]$.

Compute and store the values $\alpha^2, \alpha^3, ..., \alpha^{d_y}$; .... $O(d_y)$
$v_j \leftarrow \alpha^j a_j$ for $0 \leq j \leq d_y$.; .... $O(d_x\, d_y)$
for $k = 0$ to $d_y - 1$ **do**
  for $j = d_y - 1$ to k **do**
    $v_j \leftarrow v_j + v_{j+1}$; .... $O(d_x)$
  end
end
$v_j \leftarrow v_j / \alpha^j$ for $0 < j \leq d_y$.; .... $O(d_x\, d_y)$
return $v_0, ..., v_{d_y}$;

This idea, due to Shaw and Traub [11], has the same number of additions as Horner's method, but it needs only $d_y(d_x + 1) + d_y - 1$ multiplications and $d_y(d_x + 1)$ divisions in the field $\mathbb{F}_p$. Jong and Leeuwen in [8] show how to improve on Algorithm 9 by computing only about $\frac{1}{2}d_y$ powers of α. Performing these optimizations reduces the number of multiplications to $O(d_x d_y)$ and the number of additions to $O(d_x d_y^2)$. This means that Shaw and Traub's method for calculating the power series of A requires $O(d_x d_y^2)$ arithmetic operations. While the other algorithms (division and differentiation) have the same asymptotic cost, this algorithm's cost is bounded by addition, not multiplication, saving a significant amount of computation time.

Chapter 3

The Cubic Algorithm

3.1 Algorithm

In this chapter, we will state and calculate the complexity of our improved bivariate Hensel Lifting algorithm. Our algorithm will calculate the factorization of a polynomial $A(x, y)$ over a finite field $\mathbb{F}_p$ for some prime p by lifting $n \geq 2$ bivariate factors from univariate images. Let $d_x = \deg(A, x)$ and $d_y = \deg(A, y)$. Through two improvements we will describe throughout this chapter, we have managed to reduce the cost from Bernardin's $O(n\, d_x^2\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$ to $O(d_x^2\, d_y + d_x\, d_y^2)$. Therefore, our algorithm is cubic in respect to the degrees of d_x and d_y. We introduce a method of evaluation, point-wise multiplication in $\mathbb{F}_p$, and interpolation to asymptotically improve our algorithm by a factor of d_x or d_y, and we improve the complexity analysis of several of our sub-algorithms to reduce the overall complexity of our algorithm by an additional factor of n.

We remind the reader of what our algorithm is going to calculate. Suppose we are given some prime p and a polynomial $A \in \mathbb{F}_p[x, y]$ that is monic in x. A must also be *square-free*, that is, $\nexists\, b$ with $\deg(b, x) > 0$ such that $b^2 | A$. Suppose we pick some $\alpha \in \mathbb{F}_p$ and obtain the factorization $A(x, \alpha) = \prod_{i=1}^n f_{i,0}$ in which $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ are monic and pairwise relatively prime.

Bivariate Hensel Lifting aims to construct monic, bivariate polynomials $f_1^{(k)}, f_2^{(k)}, ..., f_n^{(k)} \in \mathbb{F}_p[x, y]$ where

$$f_i^{(k)} = f_i^{(k-1)} + f_{i,k-1}(y - \alpha)^{(k-1)} \quad \text{and} \quad f_i^{(1)} = f_{i,0}$$

such that

$$\begin{array}{ll} (1) & \forall i : f_i^{(k)} \equiv f_{i,0} \pmod{y - \alpha}, \text{ and} \\ (2) & A \equiv \prod_i f_i^{(k)} \pmod{(y - \alpha)^k}. \end{array}$$

If k is sufficiently large, the $f_i^{(k)}$ obtained can be used to compute a factorization of A over $\mathbb{F}_p$. We will stop Hensel Lifting when $\prod_i f_i^{(k)} = A(x,y)$ or when $\sum_{i=1}^{n} \deg(f_i^{(k)}, y) > \deg(A, y) = d_y$.

In Section 1.3, we showed that Bernardin's Hensel Lifting algorithm does $O(n\, d_x^2\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$. This complexity comes directly from Step 11 of Algorithm 5,

$$\Delta \leftarrow \text{CoefficientExtraction}(p, \alpha, k, f_1, f_2, ..., f_n, (y-\alpha)^k, G).$$

This process is described in detail in Section 1.3. This step refers to using convolution to multiply the n bivariate factors sequentially using univariate multiplication in $\mathbb{F}_p[x]$ to recover $\text{coeff}(\prod_{i=1}^{n} f_i^{(k)}, (y-\alpha)^k)$. We will continue to define this coefficient as $\Delta(x) \in \mathbb{F}_p[x]$. This step is the single most costly operation of Bernardin's algorithm.

To eliminate the bottleneck, we offer an alternative method to calculate $\Delta(x)$. Instead of multiplying univariate polynomials in $\mathbb{F}_p[x]$ using Karatsuba's algorithm as Bernardin did, we will use (i) polynomial evaluation in x, (ii) point-wise multiplication in $\mathbb{F}_p$, and (iii) polynomial interpolation to calculate Δ. Since we can show that $\deg(\Delta, x) < d_x$, we need d_x points $\beta_0, \ldots, \beta_{d_x-1} \in \mathbb{F}_p$ to interpolate Δ. As we will prove, the total cost of these operations is $O(d_x^2\, d_y), O(d_x\, d_y^2)$, and $O(d_x^2\, d_y)$ arithmetic operations in $\mathbb{F}_p$ respectively. Our method of calculating $\Delta(x)$ is presented as a homomorphism diagram in Figure 3.1.

$$\begin{tikzcd}
f_1^{(k)}, f_2^{(k)}, ..., f_n^{(k)} \in \mathbb{F}_p[x,y] \arrow[rr, "\text{coeff}\left(\prod_{i=1}^{n} f_i^{(k)},\, (y-\alpha)^k\right)"] \arrow[d, swap, "\substack{\text{evaluate } f_i^{(k)} \\ \text{for } 0\leq j\leq d_x-1}"] \arrow[d, "\substack{\text{at } x=\beta_j \\ \text{for } 1\leq i\leq n}"] & & \Delta \in \mathbb{F}_p[x] \\
f_{1,j}^{(k)}, f_{2,j}^{(k)}, ..., f_{n,j}^{(k)} \in \mathbb{F}_p[y] \arrow[rr, "\text{coeff}\left(\prod_{i=1}^{n} f_{i,j}^{(k)},\, (y-\alpha)^k\right)", "\text{for } 0\leq j\leq d_x-1"'] & & \Delta_j \in \mathbb{F}_p \arrow[u, "\substack{\text{interpolate } x \\ \text{Solve } \Delta(\beta_j)=\Delta_j}", "\text{for } 0\leq j\leq d_x-1"']
\end{tikzcd}$$

Figure 3.1: Homomorphism diagram for computing $\Delta(x)$ at iteration $k \geq 1$

We need to discuss the three algorithms we will use to calculate Δ. For polynomial evaluation we will use Horner's method and for interpolation we will use Lagrange interpolation. Both of these are well-known algorithms, and we discussed them in Sections 2.1 and 2.3 respectively. We need $p \geq d_x$

to have enough evaluation points to interpolate Δ. We still need to discuss the final sub-algorithm, point-wise multiplication in $\mathbb{F}_p$. The method is similar to how Bernardin used convolution and univariate multiplication to calculate $\Delta(x)$ in $\mathbb{F}_p[x]$. He found $\Delta = \text{coeff}(\prod_{i=1}^{n} f_i^{(k)}, (y-\alpha)^k)$ by saving intermediate calculations thus minimizing the multiplications needed. We will use convolution and point-wise multiplication to find the constant coefficient,

$$\Delta_j = \text{coeff}\left(\prod_{i=1}^{n} f_i^{(k)}(\beta_j, y), (y-\alpha)^k\right) \text{ for } 0 \le j \le d_x - 1.$$

We will use Bernardin's CoefficientExtraction algorithm described in Section 1.3 to accomplish this. We need to modify his algorithm so that the multiplications are done in $\mathbb{F}_p$ instead of $\mathbb{F}_p[x]$, but our algorithm uses the same method. Our algorithm is the same CoefficientExtraction algorithm where the polynomials are evaluated at $x = \beta_j$ for $0 \le j < d_x$. We still need to store the sub-products calculated as a result of the convolutions in a coefficient matrix G. However, to calculate every Δ_j, we need to use the CoefficientExtraction algorithm d_x times, so G must now be a $d_x \times n \times (d_y + 1)$ matrix of elements in $\mathbb{F}_p$, as opposed to an $n \times (d_y + 1)$ matrix of polynomials in $\mathbb{F}_p[x]$. We define an element of G as

$$G_{j,i,k} = \text{coeff}\left(\prod_{m=1}^{i} f_m^{(k+1)}(\beta_j, y), (y-\alpha)^k\right). \tag{3.1}$$

For simplicity, in our CoefficientExtraction algorithm, we will store and access the sub-products in a matrix H of size $n \times (d_y + 1)$, which is a submatrix of G that contains all the intermediate convolution calculations for a particular β_j. We will denote this submatrix as $H = G_j$ for $0 \le j \le d_x - 1$. We present our CoefficientExtraction algorithm as Algorithm 10.

Algorithm 10: Our CoefficientExtraction algorithm

Input: prime p, $\alpha \in \mathbb{F}_p$, $k \in \mathbb{Z}^+$, $f_1, f_2, ..., f_n \in \mathbb{F}_p[y]$, H an $n \times (d_y + 1)$ matrix of elements in $\mathbb{F}_p$.

Output: $\Delta = \text{coeff}(\prod_{j=1}^{n} f_j, (y-\alpha)^k) \in \mathbb{F}_p$, H an $n \times (d_y+1)$ matrix of elements in $\mathbb{F}_p$.

if $n = 2$ **then**
- $MIN \leftarrow \max(0, k - \deg(f_2, y))$;
- $MAX \leftarrow \min(k, \deg(f_1, y))$;
- $\Delta \leftarrow \sum_{j=MIN}^{MAX} \text{coeff}(f_1, (y-\alpha)^j) \cdot \text{coeff}(f_2, (y-\alpha)^{k-j})$; .... $O(k)$

else
- $d \leftarrow \deg(f_1, y)$;
- $H_{1,k} \leftarrow \text{coeff}(f_1, (y-\alpha)^k)$;
- **for** i from 2 to n **do**
  - $\delta \leftarrow d$;
  - $d \leftarrow d + \deg(f_i, y)$;
  - **if** $k \leq d$ **then**
    - $MIN \leftarrow \max(0, k - \delta)$;
    - $MAX \leftarrow \min(k, \deg(f_i, y))$;
    - $H_{i,k} \leftarrow \sum_{j=MIN}^{MAX} H_{i-1,k-j} \times \text{coeff}(f_i, (y-\alpha)^j)$; .... $O(k)$
  - **end**
- **end**
- $\Delta \leftarrow H_{n,k}$;

end

return Δ, H;

We will cover the cost of Algorithm 10 after stating our Hensel Lifting algorithm in Section 3.2.5. After we calculate $\Delta_0, ..., \Delta_{d_x - 1}$, we perform Lagrange interpolation to recover $\Delta(x) \in \mathbb{F}_p[x]$. We will then solve the multi-Diophantine equation

$$\sum_{i=1}^{n} \sigma_i \frac{\prod_{j=1}^{n} f_{j,0}}{f_{i,0}} = c_k$$

for $\sigma_1, ..., \sigma_n \in \mathbb{F}_p[x]$. We use the lifting coefficients σ_i to lift the n factors from $f_i^{(k)}$ to $f_i^{(k+1)}$ using $f_i^{(k+1)} = f_i^{(k)} + \sigma_i (y-\alpha)^k$ for $1 \leq i \leq n$.

Finally, we have to consider the problem of updating the coefficients stored in G. The same problem that occurred in Bernardin's algorithm occurs in ours, where we have to update the elements in G after lifting the factors from $f_i^{(k)}$ to $f_i^{(k+1)}$ to account for the additional terms added.

We introduce our version of Bernardin's CoefficientUpdate algorithm, which adds the necessary subproducts to G using minimal arithmetic operations. We present the CoefficientUpdate algorithm as Algorithm 11.

Algorithm 11: Our CoefficientUpdate Algorithm

Input: prime p, $\alpha \in \mathbb{F}_p$, $k \in \mathbb{Z}^+$, $f_1, f_2, ..., f_n \in \mathbb{F}_p[y]$, H an $n \times (d_y + 1)$ matrix of elements in $\mathbb{F}_p$.
Output: H an $n \times (d_y + 1)$ matrix of elements in $\mathbb{F}_p$.

if $n > 2$ **then**
  $t \leftarrow \text{coeff}(f_1, (y-\alpha)^k)$;
  $H_{1,k} \leftarrow t$;
  for i from 2 to n **do**
    $//t = f_{i,0} \sum_{j=1}^{i-1} f_{j,k} \prod_{\substack{j=m=1 \\ m \neq j}}^{i-1} f_{m,0} + f_{i,k} \prod_{j=1}^{i-1} f_{j,0}$
    $t \leftarrow \text{coeff}(f_i, (y-\alpha)^0) \times t + \text{coeff}(f_i, (y-\alpha)^k) \times H_{i-1,0}$; . . . . $O(1)$
    $H_{i,k} \leftarrow H_{i,k} + t$; . . . . $O(1)$
  end
end
return H;

We will calculate the number of arithmetic operations in Algorithm 11 in Section 3.2.5.

Example 3.1. *We demonstrate one iteration of the CoefficientExtraction and CoefficientUpdate algorithms to calculate* Δ *for the* $k = 3$ *iteration of our algorithm and store the necessary coefficients in the matrix* H*. Let* $p = 11, \alpha = 3, n = 3$*, and the field be* $\mathbb{F}_p$*. We will find* $\text{coeff}(f_1 \times f_2 \times f_3, (y-3)^3)$ *where*

$$f_1^{(3)} = -2(y-3)^2 + 2(y-3),$$

$$f_2^{(3)} = 5(y-3)^2 - 5(y-3) - 1, \text{ and}$$

$$f_3^{(3)} = -2(y-3)^2 + 2(y-3) - 4.$$

The $n \times (d_y + 1)$ *matrix* H *was input as a* 3×10 *matrix, where the first 4 columns can be seen in Table 3.1. The polynomials chosen for this example correspond to the initial polynomials in Example 1.16 evaluated at* $x = 2$*.*

n \ d_y	0	1	2	3
1	0	2	-2	-
2	0	-2	3	-
3	0	-3	-5	-

Table 3.1: The first 4 columns of matrix H which contains the intermediate products of calculating $f_1 \times f_2 \times f_3$

Following Algorithm 3, we calculate the 3rd row of H, $H_{1,3}, H_{2,3}$, and $H_{3,3}$. First, we define $H_{1,3} =\mathrm{coeff}(f_1, (y-3)^3) = 0$. *Next, we calculate $H_{2,3}$ and $H_{3,3}$ as*

$$
\begin{aligned}
H_{2,3} &= \sum_{j=1}^{2} H_{1,3-j} \times \mathrm{coeff}(f_2, (y-3)^j) \\
&= H_{1,2} \times \mathrm{coeff}(f_2, (y-3)^1) + H_{1,1} \times \mathrm{coeff}(f_2, (y-3)^2) \\
&= \mathrm{coeff}(f_1, (y-3)^2) \times \mathrm{coeff}(f_2, (y-3)^1) + \mathrm{coeff}(f_1, (y-3)^1) \times \mathrm{coeff}(f_2, (y-3)^2) \\
&= (-2) \times (-5) + (2) \times (5) \\
&= -2
\end{aligned}
$$

$$
\begin{aligned}
H_{3,3} &= \sum_{j=0}^{2} H_{2,3-j} \cdot \mathrm{coeff}(f_3, (y-3)^j) \\
&= H_{2,3} \cdot \mathrm{coeff}(f_3, (y-3)^0) + H_{2,2} \cdot \mathrm{coeff}(f_3, (y-3)^1) + H_{2,1} \cdot \mathrm{coeff}(f_3, (y-3)^2) \\
&= \mathrm{coeff}(f_1 \times f_2, (y-3)^3) \times \mathrm{coeff}(f_3, (y-3)^0) + \mathrm{coeff}(f_1 \times f_2, (y-3)^2) \times \mathrm{coeff}(f_3, (y-3)^1) \\
&\quad + \mathrm{coeff}(f_1 \times f_2, (y-3)^1) \times \mathrm{coeff}(f_3, (y-3)^2) \\
&= -2 \times -4 + 3 \times 2 + -2 \times -2 \\
&= -4
\end{aligned}
$$

So, we have updated the 4th column of H as

3
0
-2
-4

At this point of Bernardin's algorithm, we would solve the Diophantine equation, then lift the factors f_1, f_2 and f_3. In this example, we lift them to

$$f_1^{(3)} = 4(y-3)^3 - 2(y-3)^2 + 2(y-3),$$

$$f_2^{(3)} = 3(y-3)^3 + 5(y-3)^2 - 5(y-3) - 1, \textit{ and}$$

$$f_3^{(3)} = 2(y-3)^3 - 2(y-3)^2 + 2(y-3) - 4.$$

So, now we need to add the value of $f_{1,3}f_{2,0}f_{2,0} + f_{1,0}f_{2,3}f_{2,0} + f_{1,0}f_{2,0}f_{2,3}$ to H. We do this by applying the CoefficientUpdate algorithm. By executing Algorithm 4, we get

$$t = \text{coeff}(f_1, (y-3)^3) = 4$$

$$H_{3,1} = t = 4$$

$$t = \text{coeff}(f_2, (y-3)^0) \cdot t + \text{coeff}(f_2, (y-3)^3) \cdot H_{1,0} = (-1) \cdot 4 + 3 \cdot 0 = -4$$

$$H_{3,2} = H_{3,2} + t = (-2) + (-4) = 5$$

$$t = \text{coeff}(f_3, (y-3)^0) \cdot t + \text{coeff}(f_3, (y-3)^3) \cdot H_{2,0} = (-4) \cdot (-4) + 2 \cdot 0 = 5$$

$$H_{3,3} = H_{3,3} + t = (-4) + 5 = 1$$

This leads to the updated table H represented by Table 3.2.

n \ d_y	0	1	2	3
1	0	2	−2	4
2	0	−2	3	5
3	0	−3	−5	1

Table 3.2: The first 4 rows of matrix H which contains the intermediate products of calculating $f_1 \times f_2 \times f_3$

Note the polynomial in $H_{3,3}$ in Table 3.1 is equivalent to $\text{coeff}(f_1^{(3)} \times f_2^{(3)} \times f_3^{(3)}, (y-3)^3)$, *while the polynomial $H_{3,3}$ in Table 3.2 is equivalent to* $\text{coeff}(f_1^{(4)} \times f_2^{(4)} \times f_3^{(4)}, (y-3)^3)$.

△

The second improvement to Bernardin's algorithm is an improvement to the analysis of the sub-algorithms. In the initial analysis of our algorithm, we had found the complexity of polynomial evaluation, point-wise multiplication, and solving the multi-Diophantine equations to be $O(n\, d_x^2\, d_y)$, $O(n\, d_x\, d_y^2)$, and $O(n\, d_x^2\, d_y)$ respectively. However, after considering the degree bounds of each of the n factors, it is clear that the following bounds exist:

- $\sum_{i=1}^{n} \deg(f_i, x) = d_x$
- $\sum_{i=1}^{n} \deg(f_i, y) = d_y$
- $n \leq d_x$ as $\deg(f_i^{(k)}, x) \geq 1$
- $n \leq d_y$ as $\deg(f_i^{(k)}, y) \geq 1$

Using these obvious bounds, we performed a more accurate analysis of the subroutines in our algorithm and managed to remove a factor of n from the three sub-algorithms mentioned above. We will show, in Section 3.2, that performing the optimal analysis improves the overall cost of our algorithm from $O(n\, d_x^2\, d_y + n\, d_x\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$ to $O(d_x^2\, d_y + d_x\, d_y^2)$. The optimization techniques we used could have been applied to reduce the cost of Bernardin's algorithm from $O(n\, d_x^2\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$ to $O(d_x^2\, d_y^2)$. We present our new bivariate Hensel Lifting algorithm as Algorithm 12. The number of arithmetic operations done by each step is stated on the right.

We present Maple code for our bivariate Hensel Lifting algorithm in Figures A.1 and A.2 in Appendix A. We include Maple code for the main subroutines of our algorithm: the code for our CoefficientExtraction algorithm, the code for our CoefficientUpdate algorithm, the code for generating the M_i and s_i polynomials used for solving the Diophantine equations, and the code for finding a solution to the Diophantine equations, in Figures A.3, A.4, A.5, and A.6 respectively.

3.2 Analysis

In this section, we will calculate the number of arithmetic operations in $\mathbb{F}_p$ used by our improved bivariate Hensel Lifting algorithm. We restate the cost of our algorithm as Theorem 1.2.

Theorem 1.2. *Let $A \in \mathbb{F}_p[x, y], d_x = \deg(A, x) > 1, d_y = \deg(A, y) > 1$. Suppose $A = f_1 f_2 \ldots f_n$ and we are given pairwise relatively prime images $f_i(x, \alpha)$ of the factors f_i for some $\alpha \in \mathbb{F}_p$. If $p \geq d_x$, we can compute $f_1 f_2 \ldots f_n$ in $O(d_x^2\, d_y + d_x\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$ using space for $O(n\, d_x\, d_y)$ elements of $\mathbb{F}_p$.*

We will prove that the cost of our bivariate Hensel Lifting algorithm is $O(d_x^2\, d_y + d_x\, d_y^2)$ by analyzing the number of arithmetic operations done by every operation of Algorithm 12 and show that every subroutine does at most $O(d_x^2\, d_y)$ or $O(d_x\, d_y^2)$ arithmetic operations. The six subroutines we will analyze are:

1. Generating Lagrange basis polynomials - Step 4
2. Performing a base conversion on polynomial $A(x, y)$ from y to $(y - \alpha)$ - Step 5
3. Univariate polynomial evaluation - Steps 10 and 30
4. CoefficientExtraction and CoefficientUpdate - Steps 19 and 34
5. Lagrange interpolation - Step 21
6. Solving the multi-Diophantine polynomial equation - Step 25

Algorithm 12: Our Cubic Bivariate Hensel Lifting Algorithm

Input: prime p, $\alpha \in \mathbb{F}_p$, $A \in \mathbb{F}_p[x, y]$ and $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ satisfying $(i) A, f_{1,0}, f_{2,0}, ..., f_{n,0}$ are monic in x, $(ii) A(y = \alpha) = f_{1,0} f_{2,0} ... f_{n,0}$ and $(iii) \gcd(f_{i,0}, f_{j,0}) = 1$ for $i \neq j$.

Output: $f_1, f_2, ..., f_n \in \mathbb{F}_p[x, y]$ such that $A = f_1\, f_2 \, ... \, f_n$ or FAIL.

$d_x \leftarrow \deg(A, x)$; $d_y \leftarrow \deg(A, y)$;

Generate Lagrange Polynomials $L_i(x)$ s.t. $L_i(x) = \frac{\prod_{j=0}^{d_x-1}(x-j)}{(x-i)}$; .... $O(d_x^2)$

Compute $a_0, a_1, ..., a_{d_y} \in \mathbb{F}_p[x]$ s.t. $A = \sum_{k=0}^{d_y} a_k (y-\alpha)^k$; .... $O(d_x\, d_y^2)$

for $i = 1$ to n **do**
- $f_i \leftarrow f_{i,0}$;
- $df_i \leftarrow 0$; //$\deg(f_i, y)$
- $f_{i,j} \leftarrow f_{i,0}(x = j) \in \mathbb{F}_p$ for $0 \leq j \leq d_x - 1$; .... $O(\deg(f_i, x))$

end

for $j = 0$ to $d_x - 1$ **do**
- $G_{j,n,0} \leftarrow f_{n,j}$;
- **for** $i = n - 1$ by -1 to 2 **do**
  - $G_{j,i,0} \leftarrow G_{j,i+1,0} \cdot \mathrm{coeff}(f_{i,j}, (y-\alpha)^0)$; .... $O(1)$
- **end**

end

for $k = 1$ to d_y **do**
- **for** $j = 0$ to $d_x - 1$ **do**
  - $\Delta_j, G_j \leftarrow$ CoefficientExtraction$(p, \alpha, k, f_{1,j}, f_{2,j}, ..., f_{n,j}, G_j)$; .... $O(k\, n)$
- **end**
- interpolate $\Delta(x) \in \mathbb{F}_p[x]$ s.t. $\Delta(j) = \Delta_j$; .... $O(d_x^2)$
- $c_k \leftarrow a_k - \Delta$;
- **if** $\sum_{i=1}^{n} df_i = d_y$ and $c_k \neq 0$ **then Return** FAIL **end**;
- **if** $c_k \neq 0$ **then**
  - Solve $\sum_{i=1}^{n} \sigma_i \frac{f_{1,0} f_{2,0} \cdots f_{n,0}}{f_{i,0}} = c_k$ for $\sigma_1, \sigma_2, ..., \sigma_n \in \mathbb{F}_p[x]$ with $\deg(\sigma_i, x) < \deg(f_{i,0}, x)$ or $\sigma_i = 0$; .... $O(d_x^2)$
  - **for** *i=1* to n **do**
    - $f_i \leftarrow f_i + \sigma_i (y-\alpha)^k$;
    - $df_i \leftarrow \deg(f_i, y)$;
    - **for** $j = 0$ to $d_x - 1$ **do**
      - $\sigma_{ij} \leftarrow \sigma_i(x = j) \in \mathbb{F}_p$; .... $O(\deg(f_i, x))$
      - $f_{i,j} \leftarrow f_{i,j} + \sigma_{ij} (y-\alpha)^k$;
    - **end**
  - **end**
  - $G_j \leftarrow$CoefficientUpdate$(p, \alpha, k, f_{1,j}, f_{2,j}, ..., f_{n,j}, G_j)$ for $0 \leq j \leq d_x - 1$; ....$O(n\, d_x)$
- **end**

end

if $\sum_{i=1}^{n} df_i = \deg(A, y)$ **then return** $f_1, f_2, ..., f_n$ **else return** FAIL **end**;

3.2.1 Evaluation points

Before we can discuss polynomial evaluation, point-wise multiplication, and Lagrange interpolation, we need to discuss the number of evaluation points that are required to successfully recover a polynomial using interpolation. To perform Lagrange interpolation and recover $\Delta(x)$, we need $\deg(\Delta, x) + 1$ evaluation points. As we know $\deg(\Delta, x) \leq \deg(A, x)$, we could set the number of evaluation points to be $d_x + 1$ and we would always have enough points to recover Δ. However, since we are working with monic polynomials in x, this implies that if $A = \sum_{i=0}^{d_y} a_i(y-\alpha)^i$ in which $a_i \in \mathbb{F}_p[x]$, then $\deg(a_i, x) < d_x$ for $(1 \leq i \leq d_y)$ or else A is not monic in x. Therefore, $\deg(\Delta, x) < d_x$ and we require only d_x evaluation points. For simplicity, we shall let these points be

$$\{\beta_j = j : 0 \leq j \leq d_x - 1\}.$$

3.2.2 Generating Lagrange basis polynomials

We begin our analysis by considering how we generate the Lagrange basis polynomials in Step 4 of our algorithm. We described how to calculate the Lagrange basis polynomials in Section 2.3. Using the evaluation points $\beta_0, ..., \beta_{d_x-1} \in \mathbb{F}_p$, we will calculate the d_x Lagrange basis polynomials by first calculating

$$M(x) = \prod_{j=0}^{d_x-1} (x - \beta_j).$$

We then calculate the basis polynomials by dividing $M(x)$ by each of the factors

$$M_j(x) = \frac{M(x)}{(x-\beta_j)} \text{ for } 0 \leq j \leq d_x - 1.$$

Finally, we will calculate $\alpha_j = M_j(\beta_j)$ for $0 \leq j \leq d_x - 1$ using Horner's method. We then update the Lagrange basis polynomials by calculating $L_j = M_j/\alpha_j \in \mathbb{F}_p[x]$ for $0 \leq j \leq d_x - 1$. This results in the Lagrange basis polynomial having the property that $L_j(\beta_i) = 0$ if $i \neq j$ and $L_j(\beta_i) = 1$ if $i = j$.

As stated Section 2.3, the cost of generating the Lagrange basis polynomials is $O(d^2)$ where d is the number of evaluation points. Therefore, the cost of generating the Lagrange basis polynomials is $O(d_x^2)$. As the Lagrange basis polynomials do not change throughout the algorithm, we only need to generate them once at the beginning of Algorithm 12.

3.2.3 Base conversion

We must perform a base conversion on our initial polynomial $A \in \mathbb{F}_p[x,y]$ from y to $(y-\alpha)$. Using Shaw and Traub's method [20] (see Section 2.4) uses $O(d_x d_y^2)$ arithmetic operations in $\mathbb{F}_p$.

3.2.4 Polynomial evaluation

We consider the number of arithmetic operations needed to evaluate the n factors at the d_x evaluation points. First, consider the standard form of a polynomial $a(x) = a_0 + a_1 x + a_2 x^2 + ... + a_d x^d$. Horner's method, as described in Section 2.1, evaluates $a(x)$ at some value $\alpha \in \mathbb{F}_p$ using d multiplications and d additions.

In our algorithm, we perform univariate polynomial evaluation in terms of variable x on the bivariate polynomials. Let $d_{x,i} = \deg(f_i, x)$ and $d_{y,i} = \deg(f_i, y)$. Each of the n factors can be written as $f_i = \sum_{j=0}^{d_{y,i}} f_{i,j}(y-\alpha)^j$ where $f_{i,j} \in \mathbb{F}_p[x]$ and $\deg(f_{i,j}, x) \leq d_{x,i}$. We evaluate each of the factors f_i at the d_x evaluation points.

Using Horner's method, the number of multiplications and additions used to evaluate each of the n factors for a particular evaluation point β_j in our algorithm is

$$(d_{y,i}+1)(d_{x,i}).$$

Therefore, the number of arithmetic operations for evaluating the n factors at one evaluation point β_j is

$$\begin{aligned} \sum_{i=1}^{n} O((d_{y,i}+1)d_{x,i}) &= O\left(\sum_{i=1}^{n} d_{y,i} d_{x,i}\right) \subset O\left(\sum_{i=1}^{n} d_y d_{x,i}\right) \\ &= O\left(d_y \sum_{i=1}^{n} d_{x,i}\right) = O(d_x\, d_y). \end{aligned} \tag{3.2}$$

Finally, we have to evaluate each of the n factors at d_x evaluation points, so that brings the total number of arithmetic operations in $\mathbb{F}_p$ to $d_x\, O(d_x d_y) = O(d_x^2 d_y)$.

3.2.5 CoefficientExtraction and CoefficientUpdate

The next step of our algorithm is to perform the CoefficientExtraction algorithm. In the kth iteration of the main loop, it finds the coefficient of the $(y-\alpha)^k$ term from the product $f_1^{(k)}(\beta_j, y) \times f_2^{(k)}(\beta_j, y) \times ... \times f_n^{(k)}(\beta_j, y) \in \mathbb{F}_p[y]$ for one evaluation point β_j. The coefficient of the $(y-\alpha)^k$ term,

Δ_j, is needed to interpolate $\Delta(x)$ using Lagrange interpolation. We described the CoefficientExtraction algorithm earlier for Bernardin's algorithm in Section 1.3 and we covered the necessary modifications for our CoefficientExtraction algorithm in Section 3.1. We defined our CoefficientExtraction algorithm as Algorithm 10.

We consider the number of arithmetic operation in $\mathbb{F}_p$ performed by the CoefficientExtraction algorithm for the $n > 2$ case. Suppose $\deg(f_i, y) = d_y/n$ for $1 \leq i \leq n$, which maximizes the cost of Step 16 of Algorithm 10. If we are trying to maximize the number of multiplications and additions done in one iteration, then we will ignore the conditional statement in Step 13 and apply the maximum difference between the MIN and MAX variables. In any iteration k, the largest difference occurs when $MIN = 0$ and $MAX = k$. The upper bound for k is when $k = d_y/n$. In this case, our CoefficientExtraction algorithm does

$$\begin{aligned}\sum_{i=1}^{n-1}(k+1) &\leq \sum_{i=1}^{n-1}\left(\frac{d_y}{n}+1\right) < \sum_{i=1}^{n}\left(\frac{d_y}{n}+1\right)\\ &= \sum_{i=1}^{n}\frac{d_y}{n} + \sum_{i=1}^{n} 1 = d_y + n \\ &\leq 2d_y \text{ as } n \leq d_y\end{aligned} \tag{3.3}$$

multiplications every time it is used. Therefore, this algorithm has a cost of $O(d_y)$. As this algorithm is called d_y times for each of the d_x evaluation points, this algorithm has a total cost of $d_y\, d_x\, O(d_y) = O(d_x\, d_y^2)$.

We will now calculate the cost of the CoefficientUpdate algorithm. This algorithm is presented as Algorithm 11. It is easy to observe that this algorithm does exactly $2(n-1)$ multiplications and $2(n-1)$ additions with each call of the algorithm, so this algorithm has an algebraic complexity of $O(n)$. As we must use this algorithm in each of the d_y iterations of the main loop for each of the d_x evaluation points of Algorithm 12, the total cost is $d_y\, d_x\, O(n) = O(n\, d_x\, d_y)$. Since $n \leq d_y$, this algorithm uses $O(d_x\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$. Therefore, the CoefficientExtraction and CoefficientUpdate algorithms use $O(d_x\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$.

3.2.6 Lagrange interpolation

We use Lagrange interpolation to recover the polynomial $\Delta(x)$ from the data points $\{(\beta_0, \Delta_0), ..., (\beta_{d_x-1}, \Delta_{d_x-1})\}$. That is, we want to find a polynomial $\Delta(x) \in \mathbb{F}_p[x]$ such that $\Delta(\beta_j) = \Delta_j$ for $0 \leq j \leq d_x - 1$. We covered Lagrange interpolation in Section 2.3 and the cost of generating the Lagrange basis polynomials for our algorithm in Section 3.2.2. As a reminder, the Lagrange basis polynomial, denoted L_i, are d_x polynomials of degree $d_x - 1$ that have the property that $L_j(\beta_i) = 0$ if $i \neq j$ and $L_j(\beta_i) = 1$ if $i = j$ for $0 \leq i, j \leq d_x - 1$.

Given d_x Lagrange polynomials, we need to calculate $\Delta = \sum_{j=0}^{d_x-1} \Delta_j L_j(x)$ where $\Delta(\beta_j) = \Delta_j$. We calculated Δ_j by evaluating the n factors at $x = \beta_j$, then found the coefficient for the $(y-\alpha)^k$ term of the product $f_1(\beta_j, y) \times f_2(\beta_j, y) \times ... \times f_n(\beta_j, y)$. We then calculate Δ using the following matrix vector multiplication.

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{d_x-1} \end{bmatrix} = \begin{bmatrix} | & | & | & & | & | \\ L_0 & L_1 & L_2 & \dots & L_{d_x-2} & L_{d_x-1} \\ | & | & | & & | & | \end{bmatrix} \begin{bmatrix} \Delta_0 \\ \Delta_1 \\ \Delta_2 \\ \vdots \\ \Delta_{d_x-2} \\ \Delta_{d_x-1} \end{bmatrix} = L\,\Delta$$

for $c_0, ..., c_{d_x-1} \in \mathbb{F}_p$ where $\Delta = \sum_{i=0}^{d_x-1} c_i x^i$. In our C implementation, we added several optimization to the Lagrange interpolation algorithm. These will be detailed in Section 3.4.

Since $L \in \mathbb{F}_p^{d_x \times d_x}$ and $\Delta \in \mathbb{F}_p^{d_x}$, this does d_x^2 multiplications and $d_x(d_x-1)$ additions. Therefore, Lagrange interpolation uses $O(d_x^2)$ arithmetic operations in $\mathbb{F}_p$. We will perform at most d_y Lagrange interpolations for a total cost of $d_y O(d_x^2) = O(d_x^2 d_y)$.

3.2.7 Multi-Diophantine polynomial equation

We will perform cost analysis for solving the multi-Diophantine equation in Step 25 of Algorithm 12. Suppose we are given polynomials $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ such that $\gcd(f_{i,0}, f_{j,0}) = 1$ in $\mathbb{F}_p[x]$ for all $i \neq j$. Let $U(x) = \prod_{j=1}^{n} f_{j,0}$ and $U_i(x) = \frac{U(x)}{f_{i,0}}$ for $1 \leq i \leq n$. Given some polynomial $c(x) \in \mathbb{F}_p[x]$ with $\deg(c, x) < d_x$, we need to solve the equation

$$\sigma_1 U_1 + \sigma_2 U_2 + \sigma_3 U_3 + ... + \sigma_n U_n = c \tag{3.4}$$

for $\sigma_1, \sigma_2, ..., \sigma_n \in \mathbb{F}_p[x]$ where $\deg(\sigma_i, x) < \deg(f_{i,0}, x)$ for $1 \leq i \leq n$.

We covered how to solve multi-Diophantine equations of this form in Section 2.2. We will perform the complexity analysis here. We restate Algorithm 7 which finds a solution to (3.4).

Algorithm 7: Multi-Diophantine Polynomial equation Algorithm

1 **Input**: prime p, $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ satisfying $\gcd(f_{i,0}, f_{j,0}) = 1$ in $\mathbb{F}_p[x]$ for $i \neq j$, $c \in \mathbb{F}_p[x]$.

2 **Output**: $\sigma_1, \sigma_2, ..., \sigma_n \in \mathbb{F}_p[x]$.

3 $M_n \leftarrow 1$;

4 **for** $i = n$ by -1 to 2 **do** $M_{i-1} \leftarrow M_i \times f_{i,0}$ **end**; .... $O(d_x^2)$

5 $c_1 \leftarrow c$;

6 //Solve $\sigma_i M_i + \tau_i f_{i,0} = c_i$

7 **for** $i = 1$ to $n - 1$ **do**

8   Solve $s_i M_i + t_i f_{i,0} = 1$ for $s_i, t_i \in \mathbb{F}_p[x]$ using the EEA; .... $O(d_x^2)$

9     $\sigma_i \leftarrow (c_i \cdot s_i)$ rem $f_{i,0}$; .... $O(d_x^2)$

10     $\tau_i \leftarrow (c_i - \sigma_i M_i)$ quo $f_{i,0}$; .... $O(d_x^2)$

11     $c_{i+1} \leftarrow \tau_i$;

12 **end**

13 $\sigma_n \leftarrow c_n$;

14 **return** $\sigma_1, \sigma_2, ..., \sigma_n$

We will consider the following six major operations used in Algorithm 7. In Algorithm 7, we have given the costs of the main steps on the right. We must prove these costs.

1. Calculate $M_1, M_2, ..., M_n$ in Step 4
2. Solve $s_i M_i + t_i f_{i,0} = 1$ for s, t using the EEA in Step 8
3. Calculating $c_i \cdot s_i$ in Step 9
4. Multiplying $\text{rem}(c_i \cdot s_i, f_{i,0}, x)$ in Step 9
5. Calculating $\sigma_i \cdot M_i$ in Step 10
6. Multiplying $\text{quo}(c_i - \sigma_i \cdot M_i, f_{i,0}, x)$ in Step 10

Before we can consider the cost of this algorithm, we give degree bounds for the polynomials M_i, c_i, s_i, and σ_i. First, consider the initial univariate polynomials $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$. For each factor, $f_{i,0}$, let $d_i = \deg(f_{i,0}, x)$. We have $d_i > 0$ and $\sum_{i=1}^{n} d_i = d_x$.

Consider the degree bounds for each M_i. As each of the M_i are products of the initial $n - i$ factors, and the product of all factors has degree d_x, trivially $\deg(M_i, x) < d_x$. Specifically,

$$\deg(M_i, x) = d_x - \sum_{j=1}^{i} d_j = \sum_{j=i+1}^{n} d_j < d_x \text{ for } 1 \leq i \leq n.$$

The degree bounds for the polynomials c_i, s_i, and σ_i for $1 \le i \le n$ are as follows:

- $\deg(c_i, x) < dx$ because as $A = \sum_{i=0}^{d_y} a_i(y-\alpha)^i$, then $\deg(a_i, x) < dx$ for $1 \le i \le d_y$, else A is not monic.
- $\deg(s_i, x) < \deg(f_{i,0}, x) = d_i$ as a consequence of the EEA.
- $\deg(\sigma_i, x) < \deg(f_{i,0}, x) = d_i$ as $\sigma_i = (c_i s_i)$ rem $f_{i,0}$

We can now calculate cost of solving the multi-Diophantine polynomial equation.

1 Observe, in Algorithm 12, that $f_{1,0}, f_{2,0}, ..., f_{n,0}$, the initial set of factors, remain the same for every call to this algorithm. So, we can compute the polynomials $M_1, M_2, ..., M_n$ once and all the solutions to the equations $s_i M_i + t_i f_{i,0} = 1$ once and store polynomials s_i for each call to Algorithm 7.

We will calculate the products of $M_1, M_2, ..., M_n$ first. To calculate M_i, we use classical univariate multiplication. By calculating M_{i-1}, we do $(\deg(M_i, x) + 1) \times (\deg(f_{i,0}, x) + 1)$ multiplications. Therefore, the cost can be calculated as

$$\begin{aligned}
\sum_{i=1}^{n-1} O\left((\deg(M_{n-i}, x) + 1)(\deg(f_{i,0}, x) + 1)\right) &\le \sum_{i=1}^{n} O\left(\left(\sum_{j=i+1}^{n} d_j + 1\right)(d_i + 1)\right) \\
&= \sum_{i=1}^{n} O\left(d_i \sum_{j=i+1}^{n} d_j + \sum_{j=i+1}^{n} d_j + d_i + 1\right) \\
&= O\left(\sum_{i=1}^{n} d_i \sum_{j=i+1}^{n} d_j + \sum_{i=1}^{n} \sum_{j=i+1}^{n} d_j + \sum_{i=1}^{n} d_i + \sum_{i=1}^{n} 1\right) \\
&\subset O\left(d_x \sum_{i=1}^{n} d_i + \sum_{i=1}^{n} d_x + \sum_{i=1}^{n} d_i + \sum_{i=1}^{n} 1\right) \\
&= O(d_x^2 + n d_x + d_x + n) = O(d_x^2).
\end{aligned}$$

Therefore, generating $M_1, M_2, ..., M_n$ costs $O(d_x^2)$.

2 We consider the cost of executing the Extended Euclidean Algorithm $(n-1)$ times to find a solution to $s_i M_i + t_i f_{i,0} = 1$ for s_i, t_i in $\mathbb{F}_p[x]$. If the EEA is used to find the greatest common divisor of two polynomials, A, B, then it will use $3 d_A\, d_B$ multiplications, where $d_A = \deg(A, x)$ and $d_B = \deg(B, x)$ by Theorem 15.1 in [6]. Therefore, the EEA uses $O(d_A\, d_B)$ arithmetic operations in $\mathbb{F}_p$. So, the total cost of the EEA in solving the Diophantine equation is

$$\text{Cost} \sum_{i=1}^{n-1} \text{EEA}(M_i, f_{i,0}) < \sum_{i=1}^{n} O\left(\left(\sum_{j=i+1}^{n} d_j\right) d_i\right) = \sum_{i=1}^{n} O\left(d_i \sum_{j=i+1}^{n} d_j\right)$$

$$= O\left(\sum_{i=1}^{n} d_i \sum_{j=i+1}^{n} d_j\right) = O\left(d_x \sum_{i=1}^{n} d_i\right) = O(d_x^2).$$

Therefore, finding the solution to $s_i M_i + t_i f_{i,0}$ for $1 \leq i \leq n-1$ uses $O(d_x^2)$ arithmetic operations in $\mathbb{F}_p$.

3 Now, consider the cost of calculating $c_i \cdot s_i$ in the algorithm. Again, $\deg(c_i, x) \leq d_x - 1$ and $\deg(s_i, x) < \deg(f_{i,0}, x) = d_i$. Therefore, the cost is

$$\begin{aligned}
\text{Cost}\left(\sum_{i=1}^{n-1} c_i \cdot s_i\right) &< \sum_{i=1}^{n} O((d_x - 1)(d_i - 1)) \\
&= \sum_{i=1}^{n} O(d_x d_i - d_x - d_i + 1) \\
&= O\left(d_x \sum_{i=1}^{n} d_i - \sum_{i=1}^{n} d_x - \sum_{i=1}^{n} d_i + \sum_{i=1}^{n} 1\right) \\
&= O(d_x^2 - n\, d_x - d_x + n) = O(d_x^2).
\end{aligned}$$

Therefore, the cost of computing $\sum_{i=1}^{n-1} c_i \cdot s_i$ for the $n-1$ iterations of Algorithm 7 is $O(d_x^2)$ arithmetic operations in $\mathbb{F}_p$.

4 Now we can calculate the cost doing the division $(c_i s_i) \div f_{i,0}$. The division algorithm performs at most $m(n-m+1)$ multiplications, where n is the degree of the dividend, and m is the degree of the divisor. Now the degree of the dividend is

$$\deg(c_i\, s_i, x) = \deg(c_i, x) + \deg(s_i, x) \leq (d_x - 1) + (d_i - 1) = d_x + d_i - 2.$$

So, the total cost for doing the division by $f_{i,0}$ is

$$\begin{aligned}\text{Cost}\left(\sum_{i=1}^{n-1} c_i s_i \text{ rem } f_{i,0}\right) &< \sum_{i=1}^{n} O(d_i \cdot (d_x + d_i - 2 - d_i + 1)) \\ &= \sum_{i=1}^{n} O(d_i \cdot (d_x - 1)) = \sum_{i=1}^{n} O(d_x d_i - d_i) \\ &= O\left(d_x \sum_{i=1}^{n} d_i - \sum_{i=1}^{n} d_i\right) = O(d_x^2 - d_x) = O(d_x^2).\end{aligned}$$

The cost of taking the remainders is $O(d_x^2)$ arithmetic operations in $\mathbb{F}_p$.

5 Next, we will calculate the cost of finding $\sigma_i \times M_i$. Note that $\deg(\sigma_i) \leq d_i - 1$, and $\deg(M_i, x) = \sum_{j=i+1}^{n} d_j$ for $1 \leq i \leq n$. Then the cost, using classical polynomial multiplication is

$$\begin{aligned}\text{Cost}\left(\sum_{i=1}^{n-1} (\sigma_i \cdot M_i)\right) &< \sum_{i=1}^{n} O\left((d_i - 1)\left(\left(\sum_{j=i+1}^{n} d_j\right) + 1\right)\right) \\ &= \sum_{i=1}^{n} O\left(d_i \sum_{j=i+1}^{n} d_j + d_i - \sum_{j=i+1}^{n} d_j - 1\right) \\ &= O\left(\sum_{i=1}^{n} d_i \sum_{j=i+1}^{n} d_j + \sum_{i=1}^{n} d_i - \sum_{i=1}^{n} \sum_{j=i+1}^{n} d_j - \sum_{i=1}^{n} 1\right) \\ &= O\left(d_x \sum_{i=1}^{n} d_i + \sum_{i=1}^{n} d_i - \sum_{i=1}^{n} d_x - \sum_{i=1}^{n} 1\right) \\ &= O(d_x^2 + d_x - n d_x - n) = O(d_x^2).\end{aligned}$$

So again, the cost of calculating all $\sigma_i \times M_i$ is $O(d_x^2)$ arithmetic operations in $\mathbb{F}_p$.

6 Finally, we calculate the cost taking the quotient of $c_i - \sigma_i M_i \div f_{i,0}$. As we are calculating the cost of τ_i, it is implied that $\deg(c_i - \sigma_i M_i, x) \leq d_x - 1$. This has the cost

$$\begin{aligned}\text{Cost}\left(\sum_{i=1}^{n-1} (c_i - \sigma_i M_i) \text{ quo } f_{i,0}\right) &< \sum_{i=1}^{n} O(d_i(d_x - 1 - d_i + 1)) \\ &= \sum_{i=1}^{n} O(d_i(d_x - d_i)) = \sum_{i=1}^{n} O(d_x d_i - d_i^2) \\ &= \sum_{i=1}^{n} O(d_x d_i) = O\left(d_x \sum_{i=1}^{n} d_i\right) = O(d_x^2).\end{aligned}$$

Therefore, the number of arithmetic operation in $\mathbb{F}_p$ needed to find a solution to the multi-Diophantine equation is

$$6O(d_x^2) = O(d_x^2).$$

We conclude the complexity analysis of Algorithm 7 with Theorem 3.2 which provides a precise statement about the cost of finding a solution to the multi-Diophantine equation.

Theorem 3.2. *Let p be a prime. Suppose we are given $n \geq 2$ pairwise relatively prime polynomials $f_{1,0}, f_{2,0}, ..., f_{n,0} \in \mathbb{F}_p[x]$ such that $d_x = \sum_{i=1}^{n} \deg(f_{i,0}, x)$ and a polynomial $c_k \in \mathbb{F}_p[x]$ such that $\deg(c_k, x) < d_x$. Then, we can compute polynomials $f_{1,k}, ..., f_{n,k} \in \mathbb{F}_p[x]$ such that*

$$\sum_{i=1}^{n} f_{i,k} \prod_{\substack{j=1 \\ j \neq i}}^{n} f_{j,0} = c_k$$

and $\deg(f_{i,k}, x) < \deg(f_{i,0}, x)$ for $1 \leq i \leq n$ using $O(d_x^2)$ arithmetic operations in $\mathbb{F}_p$.

As we solve at most d_y Diophantine equations in Algorithm 12, they have a total cost of $d_y\, O(d_x^2) = O(d_x^2 d_y)$. We present our Maple code for generating the M_i and s_i polynomials as Figure A.5 in Appendix A and present our Maple code for solving the multi-Diophantine equation as figure A.6 in Appendix A.

3.2.8 Total cost

We will now calculate the final cost of our algorithm. We will achieve this by adding the cost of every subroutine in algorithm 12. The costs of each of the major subroutines are as follows:

1. Generating Lagrange polynomials - $O(d_x^2)$
2. Performing a base conversion - $O(d_x d_y)$ multiplications plus $O(d_x d_y^2)$ additions
3. Univariate polynomial evaluation - $O(d_x^2 d_y)$
4. CoefficientExtraction and CoefficientUpdate - $O(d_x d_y^2)$
5. Lagrange interpolation - $O(d_x^2 d_y)$
6. Solving the multi-Diophantine polynomial equation - $O(d_x^2 d_y)$

Adding the costs, Algorithm 12 does

$$O(d_x^2 + d_x d_y + d_x^2 d_y + d_x^2 d_y + d_x d_y^2 + d_x^2 d_y + d_x^2 d_y) = O(d_x^2 d_y + d_x d_y^2)$$

arithmetic operations in $\mathbb{F}_p$. So, our algorithm is cubic in respect to the degrees of d_x and d_y. This completes the proof of Theorem 1.2.

3.3 Small finite fields and general finite fields.

Throughout our thesis, we have assumed that we are working over the finite field $\mathbb{F}_p$ for some prime p. The only restriction we have placed on p is that $p \geq d_x$ so that we can have enough evaluation points to interpolate $\Delta \in \mathbb{F}_p[x]$. This was discussed in detail in Section 3.2.6. Consider the situation when $p < d_x$. The first question is: what should we do if $p < d_x$? The second question is: what if the input field is $\mathbb{F}_q$, where $\mathbb{F}_q$ is a general finite field with q elements. Does our algorithm still apply?

First, we will address whether our algorithm works over a field $\mathbb{F}_q$ where q is a prime power. As $\mathbb{F}_q$ is a field, it has the same properties as $\mathbb{F}_p$, such as being closed under addition, closed under multiplication, and multiplicative inverses exist for every non-zero element in $\mathbb{F}_q$. Therefore, all of the major operations in our algorithm, evaluation, interpolation, point-wise multiplication, solving the Multi-Diophantine equation, and performing a base conversion will work in the field $\mathbb{F}_q$. That is, Algorithm 12 works over the finite field $\mathbb{F}_q$ again provided $q \geq d_x$.

We consider the case when the field $\mathbb{F}_p$ (or $\mathbb{F}_q$) doesn't contain enough elements to perform interpolation. Let $r = p^m$ where m is the smallest positive integer such that $p^m \geq d_x$ and let $f(z) \in \mathbb{F}_p[z]$ be a monic, irreducible polynomial with a degree of m. Such a polynomial exists by Corollary 33.11 of [5]. Then $\mathbb{F}_r = \mathbb{F}_p/\langle f(z)\rangle$ is a finite field of order $r = p^m$ by Theorem 33.1 of [5]. By the above argument, our cubic Hensel lifting algorithm works over $\mathbb{F}_r$ and by extending our field from $\mathbb{F}_p$ to $\mathbb{F}_p/\langle f(z)\rangle$, we now have enough elements to successfully interpolate the polynomial $\Delta(x)$. Similarly, we can extend $\mathbb{F}_q$ to be large enough for our algorithm to interpolate $\Delta(x)$. This implies that our algorithm can work for very small fields by extending them to larger fields.

3.4 Optimizations for $\mathbb{F}_p[x, y]$

In the C implementation of our algorithm, we made several improvements that did not affect the asymptotic complexity, but are still worth mentioning. Most notably, we used an *accumulator* to reduce the number of divisions in $\mathbb{F}_p$ and we also used a set of $\pm$ points as our evaluation points. Both of these optimizations make significant improvements to the overall execution time of our algorithm.

In the C implementation of our algorithm, we implemented an *accumulator*. An accumulator is a method to reduce the number of times we have to perform a division by p operation in our code. Whenever we add or multiply any two elements of the field we have to do a modulo p operation to make sure the sum or product remains within the field. However, if we are doing convolutions of the form $\sum_{i=0}^{n} a_i b_i$ or $\sum_{i=0}^{n} a_i b_{n-i}$, then we can add all the products together and perform only one division by p at the end. This is assuming that the sum does not overflow the accumulator.

This is a major optimization as a division by p is much more expensive than integer multiplications on current hardware. It should be noted that we have implemented convolution in our evaluation, interpolation, and CoefficientExtraction subroutines, so this is a significant improvement.

We have implemented our code to store integers that are 64 bits in size, and we implemented a 128 bit accumulator. We limited the size of primes to be at most 31 bits. As all integers in the field will be at most 31 bits in length, the product of any two integers will be at most 62 bits in size. This means we can add at least $2^{128}/2^{62} = 2^{66}$ products together before we risk overflowing the accumulator. Implementing an accumulator drastically reduces the number of divisions for the convolutions $\sum_{i=0}^{n} a_i b_i$ and $\sum_{i=0}^{n} a_i b_{n-i}$ for $n+1$ to 1.

In Section 3.2.1, we stated that we would use $0, 1, ..., d-1$ as evaluation points for polynomial evaluation and interpolation. However, in our implementation, we used the points $0, \pm 1, \pm 2, ..., \pm \frac{d}{2}$. Consider some polynomial $c(x) = \sum_{i=0}^{d} c_i x^i$. If we evaluate $c(x)$ at a negative point, it results in a polynomial of the form $c(-x) = c_0 - c_1 x + c_2 x^2 - c_3 x^3 + ... + (-1)^d c_d x^d$. Observe that each coefficient of an odd degree has a negative sign compared to its positive counterpart. Assuming d is even, we can then write

$$c(-x) = \underbrace{c_0 + c_2 x^2 + c_4 x^4 + ... + c_d x^d}_{a(x^2)} - x \underbrace{(c_1 + c_3 x^2 + c_5 x^4 + ... + c_{d-1} x^{d-2})}_{b(x^2)}.$$

Let $c(x)$ be the polynomial we wish to interpolate and let $d = \deg(c, x)$. In what follows, we will assume that d is even; if not, we add 1 to d and use an additional evaluation point. It is easy to see how to evaluate a polynomial $c(x)$ in $\mathbb{F}_p[x]$ twice as fast using $\pm$ points. If we have already evaluated $c(\alpha) = a(\alpha^2) + \alpha b(\alpha^2)$ we can compute $c(-\alpha) = a(\alpha^2) - \alpha b(\alpha^2)$ using one further multiplication and subtraction. To also use the accumulator trick, we compute $a(\alpha^2)$ via the dot product $[a_0, a_2, a_4, ..., a_d] \cdot [1, \alpha^2, ..., \alpha^d]^T$ and $b(\alpha^2)$ via the dot product $[a_1, a_3, a_5, ..., a_{d-1}] \cdot [1, \alpha^2, ..., \alpha^{d-2}]^T$. For $\alpha = i$, the arrays, $[1, i^2, i^4, ..., i^d]$ for $i = 1, 2, ..., d/2$ are computed before the main Hensel loop so they can be reused.

Let $c(x) = \sum_{i=0}^{d} c_i x^i$ and assume we have computed $c(0)$ and $c(\pm i)$ for $1 \le i \le d/2$. We will use Lagrange interpolation to interpolate $c(x)$. Let

$$M(x) = \prod_{i=-d/2}^{d/2} (x - i) \text{ and } M_i(x) = \frac{M(x)}{(x-i)} \text{ for } -\frac{d}{2} \le i \le \frac{d}{2}.$$

We then use the method we described in Section 2.3. If $\alpha_i = M_i(i)$ for $1 \le i \le n$, then we calculate the polynomials $L_i(x) = M_i/\alpha_i$ for $1 \le i \le n$. This gives the polynomials the property that $L_i(j) = 0$ if $i \ne j$ and $L_i(j) = 1$ if $i = j$. The polynomials L_i are the Lagrange basis polynomials so we may write $c(x) = \sum_{i=-d/2}^{d/2} \Delta_i L_i(x)$ for some unique Δ_i.

We need a way to refer to the coefficients of $L_i(x)$. Let $L_i(x) = \sum_{j=0}^{d} L_{ij}x^j$ and $L_i = [L_{i0}L_{i1}L_{i2}...L_{id}]$. In matrix vector form, we can compute the coefficients c_i of $c(x)$ using

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_d \end{bmatrix} = \begin{bmatrix} | & | & | & & | & | \\ L_0 & L_1 & L_{-1} & \dots & L_{d/2} & L_{-d/2} \\ | & | & | & & | & | \end{bmatrix} \begin{bmatrix} \Delta_0 \\ \Delta_1 \\ \Delta_{-1} \\ \vdots \\ \Delta_{d/2} \\ \Delta_{-d/2} \end{bmatrix}$$

First note that $L_i(0) = 0$ for all $i \neq 0$ so $c_0 = \Delta_0 L_{00}$. Because d is even we have $L_{-1}(x) = L_i(-x)$ so we only compute $L_i(x)$ for $i \geq 0$. For even i we can compute c_i using $(\frac{d}{2}+1)\frac{d}{2}$ multiplications as follows.

$$\begin{bmatrix} c_2 \\ c_4 \\ c_6 \\ \vdots \\ c_d \end{bmatrix} = \begin{bmatrix} L_{02} & L_{12} & L_{22} & \dots & L_{\frac{d}{2}2} \\ L_{04} & L_{14} & L_{24} & \dots & L_{\frac{d}{2}4} \\ L_{06} & L_{16} & L_{26} & \dots & L_{\frac{d}{2}6} \\ \dots & \dots & \dots & \dots & \dots \\ L_{0\,d} & L_{1\,d} & L_{2\,d} & \dots & L_{\frac{d}{2}\,d} \end{bmatrix} \begin{bmatrix} \Delta_0 \\ \Delta_1 + \Delta_{-1} \\ \Delta_2 + \Delta_{-2} \\ \Delta_3 + \Delta_{-3} \\ \vdots \\ \Delta_{\frac{d}{2}} + \Delta_{-\frac{d}{2}} \end{bmatrix}$$

Similarly, for odd i we can compute c_i using

$$\begin{bmatrix} c_1 \\ c_3 \\ c_5 \\ \vdots \\ c_{d-1} \end{bmatrix} = \begin{bmatrix} L_{01} & L_{11} & L_{21} & \dots & L_{\frac{d}{2}1} \\ L_{03} & L_{13} & L_{23} & \dots & L_{\frac{d}{2}3} \\ L_{05} & L_{15} & L_{25} & \dots & L_{\frac{d}{2}5} \\ \dots & \dots & \dots & \dots & \dots \\ L_{0\,d-1} & L_{1\,d-1} & L_{2\,d-1} & \dots & L_{\frac{d}{2}\,d-1} \end{bmatrix} \begin{bmatrix} \Delta_0 \\ \Delta_1 - \Delta_{-1} \\ \Delta_2 - \Delta_{-2} \\ \Delta_3 - \Delta_{-3} \\ \vdots \\ \Delta_{\frac{d}{2}} - \Delta_{-\frac{d}{2}} \end{bmatrix}$$

Thus we can determine $c_1, c_2, ..., c_d$ using $d(\frac{d}{2}+1)$ multiplications. Crucially, if we do all these multiplications as dot products of vectors, we can use our accumulator optimization if the matrices are constructed in row major order.

We compute $L_i(x)$ for $0 \leq i \leq d/2$ by first computing $L(x)$ then dividing $L(x)$ by $x - i$ using polynomial division. The two matrices formed from $L_i(x)$ for $0 \leq i \leq d/2$ and the inverses of the factorials are computed once before the main Hensel Lifting loop and reused in the loop.

Chapter 4

Benchmarks

In the previous chapter, we showed that the complexity of our cubic algorithm is $O(d_x^2 d_y + d_x d_y^2)$ where $d_x = \deg(A, x)$ and $d_y = \deg(A, y)$. We wish to demonstrate, in practice, the following three facts: that our algorithm is an improvement on Bernardin's algorithm, our algorithm is cubic in the degrees of A in x and y, and changing the number of factors, n, doesn't affect the execution time. We have implemented both Bernardin's algorithm and our cubic algorithm in C. The timings were gathered using C's *clock()* function and converted to milliseconds. Tables 4.1 and 4.2 show that our algorithm is superior to Bernardin's and has a cubic cost. Table 4.3 shows that changing the number of factors n for a fixed input polynomial A does not affect the number of multiplications done. The timings were obtained on a workstation with 8 gigabytes of RAM and one Intel Core i5-4590 running at 3.3GHz base and 3.7Ghz turbo.

Table 4.1 shows timings for Hensel Lifting $n = 4$ factors in $\mathbb{F}_p[x, y]$ using both our algorithm and Bernardin's algorithm for $p = 2^{31} - 1$ and $d = d_x = d_y$. The factors f_1, f_2, f_3 and f_4 have the form $x^{d/4} + \sum_{i=0}^{\frac{d}{4}} (\sum_{j=0}^{\frac{d}{4}-1} c_{ij} x^j) y^i$ where the coefficients c_{ij} are chosen at random from $[0, p)$. We then input $\alpha = 3, A = f_1 \times f_2 \times f_3 \times f_4$, $f_{1,0} = f_1(x, \alpha), f_{2,0} = f_2(x, \alpha), f_{3,0} = f_3(x, \alpha)$, and $f_{4,0} = f_4(x, \alpha)$ to the Hensel Lifting algorithms.

The first column represents the degrees of the polynomial A which is being factored. It has degree d in variables x and y. The second column labelled "Time(ms)" is for the time it took for our cubic $O(d_x^2 d_y + d_x d_y^2)$ algorithm to completely execute. The third column labelled "Ratio" is computed as

$$\frac{\text{Execution time for } 2d}{\text{Execution time for } d}.$$

Eg. $4.14 = 1.49/0.36$. Similarly, we compute the "Time(ms)" and "Ratio" of Bernardin's algorithm in columns 4 and 5. The speedup is a comparison between the time it took to compute our algorithm compared to Bernardin's.

	Our Cubic Algorithm		Bernardin's Algorithm		
Deg A d	Time(ms)	Ratio	Time(ms)	Ratio	Speedup
16	0.13	-	0.11	-	0.85
32	0.36	2.77	0.43	3.91	1.19
64	1.49	4.14	3.38	7.86	2.27
128	7.77	5.21	34.40	10.18	4.43
256	45.38	5.84	362.85	10.54	8.00
512	324.19	7.14	4319.31	11.90	13.32
1024	2502.76	7.72	55716.30	12.90	22.26
2048	18017.49	7.19	782982.80	14.05	43.46
4096	128211.01	7.11	11647207.28	14.88	90.84
8192	963335.81	7.51	-	-	-

Table 4.1: Hensel Lifting timings for $\mathbb{F}_p[x, y]$ with $p = 2^{31} - 1$ and $n = 4$ factors. Compares the overall execution times of our algorithm vs Bernardin's algorithm

Our algorithm significantly outperformed Bernardin's algorithm. In the final iteration, when $d_x = d_y = 4096$, our algorithm ran approximately 91 times faster than Bernardin's. As expected, the timings of our algorithm increased a factor approaching eight, while Bernardin's increased by a factor approaching sixteen as we doubled the degrees of d_x and d_y.

Our algorithm performs slightly better than the expected $\Theta(d_x^2 d_y + d_x d_y^2)$. To illustrate that our algorithm is indeed cubic, we will count the number of multiplications performed by each of the subroutines. In Section 3.2, we proved that polynomial evaluation, interpolation, the CoefficientExtraction and CoefficientUpdate algorithms, and solving the Diophantine equation all have cubic cost. Therefore, we consider the number of multiplications done by these four subroutines.

In Table 4.2, we use the same polynomials and setup from the previous experiment. Column one refers to the degree of polynomial A which is being factored. Columns two, four, six, and eight represent the total number of multiplications in $\mathbb{F}_p$ performed by our algorithm. Columns three, five, seven, and nine, which are labeled as "Ratio", are computed as

$$\frac{\text{Number of multiplications for } 2d}{\text{Number of multiplications for } d}.$$

As we can see, the number of multiplications done by the four major subroutines of our algorithm increase by a factor of eight when doubling the degrees of d_x and d_y. This shows that our algorithm is cubic. Now we will show that changing the number of factors, n, does not change the number of multiplications done by our algorithm.

	Number of multiplications in $\mathbb{F}_p$			
$degA$ d	Evaluations	Ratio	Diophantine	Ratio
16	544	-	2,064	-
32	4,160	7.647	16,704	8.093
64	32,896	7.908	134,400	8.046
128	262,400	7.977	1,078,272	8.023
256	2,097,664	7.994	8,638,464	8.011
512	16,778,240	7.999	69,156,864	8.006
1024	134,219,776	8.000	553,451,520	8.003
2048	1,073,745,920	8.000	4,428,398,592	8.001
4096	8,589,942,784	8.000	35,430,334,464	8.001
8192	68,719,493,120	8.000	283,455,258,624	8.000
$degA$ d	CoeffExtract + CoeffUpdate	Ratio	Interpolation	Ratio
16	1,989	-	1,168	-
32	14,025	7.051	8,736	7.479
64	105,105	7.494	67,648	7.744
128	813,345	7.738	532,608	7.873
256	6,398,529	7.867	4,227,328	7.937
512	50,758,785	7.933	33,686,016	7.969
1024	404,359,425	7.966	268,960,768	7.984
2048	3,228,045,825	7.983	2,149,582,848	7.992
4096	25,797,075,969	7.992	17,188,261,888	7.996
8192	206,267,492,352	7.996	137,472,516,096	7.998

Table 4.2: Hensel Lifting timings for $\mathbb{F}_p[x,y]$ with $p = 2^{31} - 1$ and $n = 4$ factors. Compares the number of multiplications done by the polynomial evaluation, solving the Diophantine equation, coefficient extraction and coefficient update, and polynomial interpolation subroutines

In Table 4.3, we set the degrees of the bivariate polynomial $A(x,y)$ to be 2048 in x and y. The factors $f_1, ..., f_n$ have the form $x^{d/n} + \sum_{i=0}^{\frac{d}{n}}(\sum_{j=0}^{\frac{d}{n}-1} c_{ij}x^j)y^i$ where the coefficients c_{ij} are chosen at random from $[0,p)$. We then input $\alpha = 3, A = f_1 \times f_2 \times ... \times f_n, f_{1,0} = f_1(x,\alpha), ..., f_{n,0} = f_n(x,\alpha)$ into the Hensel Lifting algorithm.

The first column, "n", represents the number of factors for polynomial $A(x,y)$. The second column labelled "Time(ms)" is for the time it took for our cubic algorithm to execute. The third column labelled "Ratio" is computed as

$$\frac{\text{Execution time for } 2n}{\text{Execution time for } n}.$$

The fourth, sixth, and eighth column represent the total number of multiplications in $\mathbb{F}_p$ performed by our algorithm. The fifth, seventh, and ninth column labeled "Ratio" are computed as

$$\frac{\text{Number of multiplications for } 2n}{\text{Number of multiplications for } n}.$$

			Number of multiplications in $\mathbb{F}_p$					
n	Time(ms)	Ratio	Evaluations	Ratio	Diophantine	Ratio	CoeffExtraction + CoeffUpdate	Ratio
4	17,678.54	-	1,073,745,920	-	4,428,398,592	-	3,228,047,874	-
8	19,111.93	1.081	536,879,104	0.500	2,230,910,976	0.503	3,774,100,227	1.169
16	23,648.20	1.237	268,451,840	0.500	1,100,759,040	0.493	4,059,710,337	1.076
32	31,204.01	1.320	134,250,496	0.500	544,346,112	0.495	4,227,746,778	1.041
64	42,223.92	1.353	67,174,400	0.500	270,369,792	0.497	4,362,161,178	1.032
128	66,773.93	1.581	33,685,504	0.500	134,697,216	0.498	4,530,179,178	1.039

Table 4.3: Hensel Lifting timings for $\mathbb{F}_p[x, y]$ with $p = 2^{31} - 1$. Compares the timings and number of multiplications performed when $d_x = d_y = 2048$ for our algorithm.

We wish to demonstrate that changing the number of factors, n, does not affect the execution time of our algorithm. If you observe column three of Table 4.3, as the value of n doubles, it appears as though the execution time is increasing as opposed to remaining unchanged. This contradicts the expected results, so we need to do some additional testing.

By observing our algorithm, Algorithm 12, there are only three sub-algorithms that can be asymptotically affected by the number of factors: polynomial evaluations (Steps 9, 30), solving the multi-Diophantine equations (Step 25), and the CoefficientExtraction and CoefficientUpdate sub-algorithms (Steps 19, 34). So, instead of looking at the execution times of these operations, we will count the number of multiplications they perform during the execution of Algorithm 12. We expect that, by changing the number of n factors, the number of multiplications should not change. By observing the fifth and seventh columns of Table 4.3, we see that doubling the number of factors causes the number of multiplications to be reduced by a factor approaching one half when performing evaluations and solving the multi-Diophantine equations. We also see that, in the ninth column, the number of multiplications appears to be approaching 1.0 when we double the number of factors for the CoefficientExtraction and CoefficientUpdate algorithms. The results for the CoefficientExtraction and CoefficientUpdate algorithms were as expected, but the results for the other two sub-algorithms were not expected.

Consider the number of times that the polynomial evaluation and solving the multi-Diophantine equations sub-algorithms are called. They are only used if we are trying to find non-zero polynomials $\sigma_1, ..., \sigma_n$ to update the factors $f_i^{(k)}$ in the kth iteration. In other words, we only call those two subroutines if $c_k \neq 0$ in the kth iteration of Algorithm 12. In our experiments, we let the degree in y of each of the n factors be equal to d/n. Therefore, after d/n iterations of the main loop of our algorithm, we will have found the complete factorization of A. So, if we double n, it takes half as many iterations to find a factorization of A, and there are half as many calls to perform polynomial evaluation and solve multi-Diophantine equations. This is confirmed in the fifth and seventh columns of Table 4.3. Therefore, we have confirmed that changing the number of factors when using our algorithm doesn't affect the number of multiplications done. Therefore, the number of factors does not asymptotically affect the complexity of Algorithm 12.

This leads to the question: why does increasing the number of factors significantly increase the execution time of our algorithm? This is only speculation, but the reason for the increase in execution time is most likely the workstation accessing the data stored for our algorithm. There are several arrays in our code of size $n \times d_x \times (d_y + 1)$, including G which we use in the CoefficientExtraction algorithm. These arrays double in size as we double n, so it would take longer to go through the arrays and extract the necessary data.

Chapter 5

Conclusion

We have created a new Hensel Lifting algorithm which calculates a factorization of a monic polynomial $A(x, y)$ over a finite field $\mathbb{F}_p$ by lifting $n \geq 2$ bivariate factors from univariate images. Let A have degree d_x and d_y in variables x and y respectively. The algorithm we have created does $O({d_x}^2 d_y + d_x\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$ to Hensel lift n factors of A.

Our algorithm was based on Bernardin's algorithm[2]. We made two major improvements to his algorithm. We showed that using polynomial evaluation, interpolation, and point-wise multiplication to calculate $\Delta(x)$ uses fewer arithmetic operations than univariate polynomial multiplication. We have also shown that we can remove a factor of n from the cost of Bernardin's algorithm by minimizing the number of multiplications done by every subroutine. Both of these changes lead to an improvement from $O(n\, d_x^2\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$ to $O({d_x}^2 d_y + d_x\, d_y^2)$.

Finally, we consider the results in Chapter 4. We have shown, in practice, that our algorithm is more efficient than Bernardin's algorithm for polynomials with a degree greater than 20 in x and y. We have also confirmed, through execution time and the number of multiplications done, that our algorithm is cubic in the degree of A. Finally, we have demonstrated that while increasing the number of factors does slightly increase the number of multiplications performed by the CoefficientExtraction algorithm, it does not change the complexity.

In summary, Chapter 1 covered the classical Hensel Lifting algorithm for factoring polynomials in $\mathbb{Z}[x]$ and showed how Bernardin improved upon it to create an algorithm to factor polynomials in $\mathbb{F}_p[x, y]$ into $n \geq 2$ factors. Then, in Chapter 2, we discussed well-known algorithms that were needed in our algorithm. In Chapter 3, we outlined our improvements to Bernardin's algorithm, stated our new algorithm, and proved that our algorithm uses $O(d_x^2\, d_y + d_x\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$. Finally, in Chapter 4, we presented some benchmarks for the implementation of our algorithm in C, as well as analysis of the results. The results showed that our algorithm runs faster than Bernardin's algorithm and verifies that it uses $O(d_x^2\, d_y + d_x\, d_y^2)$ arithmetic operations in $\mathbb{F}_p$.

Throughout this thesis, we have shown that our algorithm is superior to Bernardin's algorithm in theory and in practice.

We conclude this thesis with a look at what is to come. First and foremost, we would like to investigate the case when A and the factors $f_1, ..., f_n$ are not monic in x. We started with the monic case because it is easier to work with. In [18], Monagan showed that the two factor case can be modified to work for the non-monic case. After that, we hope to find a proof for Conjecture 1.17. Finally, we hope to apply our method to improve the algorithm described by Monagan and Tuncer in [18]. They developed an algorithm to use Hensel Lifting to find a factorization of multivariate polynomials in m variables. They use evaluation to reduce the multivariate polynomial to many bivariate images, then use bivariate Hensel Lifting to factor the images, and finally use sparse interpolation to recover the factorization of the original multivariate polynomial. Their algorithm would benefit from using our algorithm to perform their bivariate lifts.

Bibliography

[1] J. L. Berggren, Sharaf Al-Dīn Al-Tīusīi, Roshdi Rashed, and Sharaf Al-Din Al-Tusi. Innovation and Tradition in Sharaf Al-dīn Al-tīsīs Mu'ādalāt. *Journal of the American Oriental Society*, **110**(2):304–309, 1990.

[2] Laurent Bernardin. On bivariate Hensel lifting and its parallelization. *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation - ISSAC 98*, pages 96–100, 1998.

[3] David G. Cantor and Hans Zassenhaus. A New Algorithm for Factoring Polynomials Over Finite Fields. *Mathematics of Computation*, **36**(154):587–587, 1981.

[4] David A. Cox, John B. Little, and Donal OShea. *Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra.* Springer, 3 edition, 2008.

[5] John B. Fraleigh. *A First Course in Abstract Algebra.* Pearson, 7 edition, 2014.

[6] Joachim von zur Gathen and Gerhard Jürgen. *Modern Computer Algebra.* Cambridge University Press, 3 edition, 2013.

[7] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic, 1992.

[8] Lieuwe De Jong and Jan Van Leeuwen. An Improved Bound on the Number of Multiplications and Divisions necessary to Evaluate a Polynomial and all its Derivatives. *ACM SIGACT News*, **7**(3):32–34, 1975.

[9] Erich Kaltofen. Sparse Hensel Lifting. *EUROCAL 85 Lecture Notes in Computer Science*, pages 4–17, 1985.

[10] Erich Kaltofen and Barry M. Trager. Computing with Polynomials Given by Black Boxes for Their Evaluations: Greatest Common Divisors, Factorization, Separation of Numerators and Denominators. *Journal of Symbolic Computation*, **9**(3):301–320, 1990.

[11] D. E. Knuth. *The Art of Computer Programming. 2.ed. 2: Seminumerical algorithms.* Addison-Wesley, 1981.

[12] Martin Mok-Don Lee and Gerhard Pfister. *Factorization of Multivariate Polynomials.* PhD thesis, Simon Fraser University, 2013.

[13] Erik Meijering. A Chronology of Interpolation: From Ancient Astronomy to Modern Signal and Image Processing. In *Proceedings of the IEEE*, pages 319–342, 2002.

[14] M. Mignotte. Some Useful Bounds. *Computing Supplementa Computer Algebra*, pages 259–263, 1983.

[15] Alfonso Miola and David Y. Y. Yun. Computational Aspects of Hensel-type Univariate Polynomial Greatest Common Divisor Algorithms. *SIGSAM Bull.*, **8**(3):46–54, August 1974.

[16] Michael Monagan. Linear Hensel Lifting for $\mathbb{Z}_p[x, y]$ and $\mathbb{Z}[x]$ with Cubic Cost. *To appear in Proceedings of the 2019 International Symposium on Symbolic and Algebraic Computation*, 2019.

[17] Michael Monagan and Baris Tuncer. Using Sparse Interpolation in Hensel Lifting. volume **9890**, pages 381–400, 09 2016.

[18] Michael Monagan and Baris Tuncer. Sparse Multivariate Hensel Lifting: A High-Performance Design and Implementation. *Mathematical Software - ICMS 2018 Lecture Notes in Computer Science*, pages 359–368, 2018.

[19] Norman R. Reilly. *Introduction to Applied Algebraic Systems*. Oxford University Press, 2010.

[20] Mary Shaw and J. F. Traub. On the Number of Multiplications for the Evaluation of a Polynomial and Some of Its Derivatives. *J. ACM*, **21**(1):161–167, January 1974.

[21] Paul Wang. An Improved Multivariate Polynomial Factoring Algorithm. *Mathematics of Computation*, **32**:1215–1231, 01 1978.

[22] Paul S. Wang. Parallel Polynomial Operations on SMPs: an Overview. *Journal of Symbolic Computation*, **21**(4-6):397–410, 1996.

[23] Paul S. Wang and Linda Preiss Rothschild. Factoring Multivariate Polynomials Over the Integers. *Mathematics of Computation*, **29**(131):935–950, 1975.

[24] Edward Waring. VII. Problems concerning Interpolations. *Philosophical Transactions of the Royal Society of London*, **69**:59–67, 1779.

[25] Hans Zassenhaus. On Hensel Factorization, I. *Journal of Number Theory*, **1**(3):291–311, 1969.

Appendix A

Code

```
> CubicBivariateHensel := proc(A::polynom,F0::list,x::name,y::name,alpha::integer,p::prime)
  #A(x,y) - polynomial to factor
  #F0 - list of monic, relatively prime
  #   polynomials in x s.t. the product is equal to...
  #x - variable 1 (usually x)
  #y - variable 2 (usually y)
  #alpha - integer to use for calculating the taylor coeff.
  #p - prime

  #local variables
  local n,B,F,E,Evals,G,i,j,k,t,ck,sigmas
  local deltas,Delta,Coeffs,CoeffsMul,evalPoints,dx,dy,T,M,S;

  n := nops(F0);
  F := F0;
  dx := degree(A,x);
  dy := degree(A,y);

  #initial arrays/lists
  deltas := [seq(0,i=1..dx)];
  evalPoints := [seq(i,i=1..dx)];

  #Solve for M polynomials to use for the Diophantine Equation (Optimization)

  T,M,S := MultiEEA(F0,x,p); # Solve this once for re-use

  #if the EEA failed
  if (T,M) = (FAIL,FAIL) then
     return FAIL;
  fi;

  #do initial evaluations
  Evals := Array(1..dx);
  for i from 1 to dx do
     Evals[i] := Array(1..n);
     for j from 1 to n do
        Evals[i][j] := Eval(F0[j],x=evalPoints[i]) mod p;
     od;
  od;

  #set up G (CoeffExtract matrix)
  G := Array(1..dx);
  for i from 1 to dx do
     G[i] := Matrix(n,dy+1);
     G[i][1,1] := Evals[i][1];
     for j from 2 to n do
        G[i][j,1] := Evals[i][j]*G[i][j-1,1] mod p;
     od;
  od;

  #get a taylor series around (y-alpha)
  #(does NOT use Shaw and Traub's method)
  B := taylor(A,y=alpha,dy+1);
```

Figure A.1: Maple Code for our Cubic Algorithm part 1

```
for k from 1 to dy do

   #CoefficientExtraction
   for i from 1 to dx do
      deltas[i],G[i] := CoeffExtract(Evals[i],G[i],p,k,y);
   od;

   #Interpolation
   Delta := Interp(evalPoints,deltas,x) mod p;

   ck := expand(coeff(B,(y-alpha),k) - Delta);

   if ck <> 0 then

       #Solve Diophantine Equation for coefficients
       sigmas := DiophantineN(F0,ck,M,S,p,x);

       #Update the values of F
       for i from 1 to n do
          F[i] := F[i] + sigmas[i]*(y-alpha)^k;
       od;

       #Update Evaluations
       for i from 1 to dx do
          for j from 1 to n do
             t := Eval(sigmas[j],x=evalPoints[i]) mod p;
             Evals[i][j] := Evals[i][j] + t*y^k;
          od;
       od;

       #Perform CoefficientUpdate
       for i from 1 to dx do
          G[i] := CoeffUpdate(Evals[i],G[i],p,k,y);
       od;
   fi;
od;

#return bivar polynomials or fail
print(F);print(dy);
if add(degree(F[i],y),i=1..n) = dy then
   return(F);
else
   return(FAIL);
fi;

end proc:
```

Figure A.2: Maple Code for our Cubic Algorithm part 2

```
> #Our CoefficientExtraction algorithm
  CoeffExtract := proc(Fn,G,p,k,y)
     #Fn - n polynomials in F_p[y]
     #H - n x dy matrix of elements in F_p
     #p - prime p
     #k - iteration counter
     #y - variable of coefficients

     local n,MIN,MAX,i,j,d,delta,Delta,H;

     # number of factors
     n := nops(Fn);
     Delta := 0;
     H := G;

     #n = 2 case
     if n = 2 then
        MIN := max(0,k-degree(Fn[2],y));
        MAX := min(k,degree(Fn[1],y));
        for j from MIN to MAX do
           Delta := Delta + coeff(Fn[1],y,j)*coeff(Fn[2],y,k-j) mod p;
        od;
     #n > 2 case
     else
        d := degree(Fn[1],y);
        H[1,k+1] := coeff(Fn[1],y,k);
        for i from 2 to n do
           delta := d;
           d := d + degree(Fn[i],y);
           if k <= d then
              MIN := max(0,k-delta);
              MAX := min(k,degree(Fn[i],y));
              for j from MIN to MAX do
                 H[i,k+1] := H[i,k+1] + H[i-1,k-j+1]*coeff(Fn[i],y,j)  mod p;
              od;
           fi;
        od;
        Delta := H[n,k+1];
     fi;

     return (Delta, H);

  end proc:
```

Figure A.3: Maple Code for the Coefficient Extraction Algorithm

```
> #Our CoefficientUpdate Algorithm
CoeffUpdate := proc(Fn,G,p,k,y)

   local n,t,H,i;

   n := nops(Fn);
   H := G;

   if n > 2 then
      t := coeff(Fn[1],y,k);
      H[1,k+1] := t;
      for i from 2 to n do
         t := coeff(Fn[i],y,0)*t + coeff(Fn[i],y,k)*H[i-1,0+1] mod p;
         H[i,k+1] := H[i,k+1] + t mod p;
      od;
   fi;
   return H;
 end proc:
```

Figure A.4: Maple Code for the Coefficient Update Algorithm

```
> #Calculates M,S polynomials for Diophantine Equations
  #Verifies the initial n input polynomials are relatively prime
  MultiEEA := proc(U::list,x::name,p::prime)
  local n,M,sis,i,g,s,t,Mpolys,Spolys;

      #local variables
      n := nops(U);
      Mpolys := Array(2..n);
      Spolys := Array(1..n-1);
      M := 1;

      #Generate M
      for i from n by -1 to 2 do
          M := Expand(M*U[i]) mod p;
          Mpolys[i] := M;
      od;
      Mpolys := convert(Mpolys,list);
      sis := NULL;

     #Generate S, verify polynomials relatively prime
      for i from 1 to n-1 do
          g := Gcdex(Mpolys[i],U[i],x,'Spolys[i]') mod p;
          if g=1 then sis := sis,s; else return FAIL,FAIL; fi;
      od;
      [sis],Mpolys,convert(Spolys,list);
  end:
```

Figure A.5: Maple Code for which generates polynomials for the Diophantine Equation

```
> #Solve the Diophantine Equation for #n factors
  #Note we pre-calculated the polynomials of M,s for re-use
  DiophantineN := proc(U,c,M,S,p,x)

  local n,q,g,ck,i,s,t,Sigmas;

  n := nops(U);
  ck := c;
  Sigmas := Array(1..n);
  for i from 1 to n-1 do
     Sigmas[i] := Rem(ck*S[i],U[i],x) mod p;
     ck := Quo(ck-Sigmas[i]*M[i],U[i],x) mod p;
  end do;
  Sigmas[n] := ck;
  return(convert(Sigmas,list));
  end proc:
```

Figure A.6: Maple Code for solving the Multi-Diophantine Equation for n factors