EFFICIENT ALGORITHMS FOR COMPUTATIONS WITH SPARSE POLYNOMIALS

by

Seyed Mohammad Mahdi Javadi
B.Sc., Sharif University of Technology, 2004
M.Sc., Simon Fraser University, 2006

a Thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Spring 2011

APPROVAL

Name: Seyed Mohammad Mahdi Javadi

Degree: Doctor of Philosophy

Title of Thesis: Efficient Algorithms for Computations with Sparse Polynomials

Examining Committee: Dr. Valentine Kabanets
Chair

____

Dr. Michael Monagan, Senior Supervisor

____

Dr. Arvind Gupta, Supervisor

____

Dr. Marni Mishna, SFU Examiner

____

Dr. Mark Giesbrecht, External Examiner

Date Approved: January 19, 2011

Declaration of Partial Copyright Licence

Abstract

The problem of interpolating a sparse polynomial has always been one of the central objects of research in the area of computer algebra. It is the key part of many algorithms such as polynomial GCD computation. We present a probabilistic algorithm to interpolate a sparse multivariate polynomial over a finite field, represented with a black box. Our algorithm modifies the Ben-Or/Tiwari algorithm from 1988 for interpolating polynomials over rings with characteristic zero to positive characteristics by doing additional probes. To interpolate a polynomial in n variables with t non-zero terms, Zippel's algorithm interpolates one variable at a time using $O(ndt)$ probes to the black box where d bounds the degree of the polynomial. Our new algorithm does $O(nt)$ probes. We provide benchmarks comparing our algorithm to Zippel's algorithm and the racing algorithm of Kaltofen/Lee. The benchmarks demonstrate that for sparse polynomials our algorithm often makes fewer probes. A key advantage in our new algorithm is, unlike the other two algorithms, it can be parallelized efficiently.

Our main application for an efficient sparse interpolation algorithm is computing GCDs of polynomials. We are especially interested in polynomials over algebraic function fields. The best GCD algorithm available is SparseModGcd, presented by Javadi and Monagan in 2006. We further improve this algorithm in three ways. First we prove that we can eliminate the trial divisions in positive characteristic. Trial divisions are the bottleneck of the algorithm for denser polynomials. Second, we give a new (and correct) solution to the normalization problem. Finally we will present a new in-place library of functions for computing GCDs of univariate polynomials over algebraic number fields.

Furthermore we present an efficient algorithm for factoring multivariate polynomials over algebraic fields with multiple field extensions and parameters. Our algorithm uses Hensel lifting and extends the EEZ algorithm of Wang which was designed for factorization over rationals. We also give a multivariate p-adic lifting algorithm which uses sparse interpolation. This enables us to avoid using poor bounds on the size of the integer coefficients in the factorization when using Hensel lifting. We provide timings demonstrating the efficiency of our algorithm.

To my beloved wife, Maryam,
and my wonderful son, Ali,
and my dearest parents, Nahid and Ahmad

بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

Acknowledgments

My foremost gratitude goes to my wonderful adviser, Michael Monagan, whose invaluable guidance and generous support has always been with me during my Ph.D. studies at SFU. It was a real pleasure for me to do my Ph.D. with him. His continuous encouragement and belief in me were vital in the advancement of my graduate career. I'm especially thankful to Mike for being constantly available for discussions that would always lead to new ideas, thoughts, and insights. For all this and more, I gratefully thank him.

I also thank the members of my thesis committee: Mark Giesbrecht, Arvind Gupta, Marni Mishna and Valentine Kabanets for reading my thesis, being present at my defence session and providing me with many feedbacks.

I also take this opportunity to thank one of my best friends at SFU, Roman Pearce. We had many, many wonderful discussions about computer algebra and most importantly life! He taught me a great deal about his fascinating work on Maple. We were on several trips together. I will never forget our several hours of walking in Seoul. Thank you Roman!

I'd like to extend my warmest gratitude to one of my best and most brilliant friends; Roozbeh Ghaffari for being so amazingly caring, kind and supportive. Roozbeh has undoubtedly been one of the most influential people in my life. Thank you Roozbeh!

Finally, I am deeply thankful to my family for all the things they have done for me. I am grateful to my parents for their unconditional support, constant encouragement, and faith in me throughout my whole life. I am also deeply thankful to my beloved wife, colleague and best friend, Maryam, whose love, help and tolerance exceeded all reasonable bounds. She made my study easier with love and patience: without her constant support and encouragement this thesis would have been an impossibility. I should also thank my lovely son, Ali who always supports me with his little heart!

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

In this thesis we are interested in the design and implementation of efficient algorithms for computations with *sparse* multivariate polynomials. These include sparse polynomial interpolation, sparse GCD computation and sparse polynomial factorization.

Definition 1.1. Let f be a polynomial in variables $x_1, \ldots, x_n$ with t non-zero terms. Let T be the total number of possible terms considering the degree bounds. The polynomial f is *sparse* if

$$\frac{t}{T} \ll 1.$$

The problem of interpolating a polynomial from its *black box* representation has been of interest for a long time. It is a key part of many algorithms in computer algebra (e.g. see [84, 31, 9]). There are various efficient algorithms (e.g. Newton's interpolation method) for the case where the target polynomial is dense. These algorithms have poor performances for sparse polynomials.

Example 1.2. For $f = x_1^d + x_2^d + \cdots + x_n^d + 1$, Newton's algorithm uses $(d+1)^n$ evaluation points even though f has only $n+1$ non-zero terms.

We are especially interested in problems with a sparse target polynomial. That is, we want to design algorithms which have time complexity polynomial in the number of terms t, the degree d and the number of variables n (rather than exponential in n and d).

The problem of computing greatest common divisors is an important tool in computer algebra systems (Maple, Mathematica, ...) with many applications including simplifying rational expressions, polynomial factorization and symbolic integration. Having an efficient algorithm for computing GCDs is one of the most important parts of any general purpose computer algebra system [58]. Although the Euclidean algorithm may be used for computing the GCDs of polynomials, it has a poor performance due to the exponential growth of the coefficients. This is especially a problem for multivariate polynomials. The solution is to use *modular algorithms* [7, 9, 71, 31]. A modular algorithm basically works modulo a prime and evaluates all the variables but one. Our goal is to have efficient algorithms for the case where the GCD is sparse. Unfortunately the GCD of two sparse polynomials may not be sparse, i.e. the GCD might have many more terms compared to the input polynomials.

Example 1.3. Let $f_1 = x^{45} + 1$ and $f_2 = x^{27} - x^{18} + x^6 - x^{21} + x^9 - 1$. The GCD $g = \gcd(f_1, f_2) = x^{24} + x^{21} - x^{15} - x^{12} - x^9 + x^3 + 1$ has more terms than both f_1 and f_2.

We are interested in algorithms which are both *input and output sensitive*, i.e. the time complexity depends on the size of the inputs and the GCD. Our work in this thesis is mainly focused on computing sparse GCDs of polynomials over an algebraic number or function field.

Polynomial factorization is one of the most successful areas in computer algebra. There are some polynomial-time complexity algorithms for factoring polynomials with integer (or rational) coefficients (e.g see [50, 72]). Factorization has many applications but especially used for solving systems of polynomial equations (See [73]). Another application of factorization is in coding theory for developing error correcting codes (See e.g. [5]). Similar to the GCD problem, some of the factors of a sparse polynomial may be dense.

Example 1.4. The factorization of the sparse polynomial $f = x^7 - y^7$ is

$$(x - y)\left(x^6 + x^5y + x^4y^2 + x^3y^3 + x^2y^4 + xy^5 + y^6\right).$$

In this thesis, we are interested in having an efficient algorithm for factoring polynomials over algebraic function fields.

In this chapter we will present the state of the art algorithms for sparse polynomial interpolation (Section 1.1), sparse polynomial GCD computation (Section 1.2) and factorization of sparse polynomials (Section 1.3).

1.1 Polynomial Interpolation

Let $\mathbb{F}$ be an arbitrary field and let $f = \sum_{i=1}^{t} C_i \times M_i \in \mathbb{F}[x_1, \ldots, x_n]$ be a polynomial in n variables with $C_i \in \mathbb{F}\backslash\{0\}$. Thus f has t non-zero terms. Here $M_i = x_1^{e_{i1}} \times \cdots \times x_n^{e_{in}}$ is the i'th monomial in f and $e_{ij} = \deg_{x_j}(M_i) \in \mathbb{Z}$. Let B be a *black box* that on input $(\alpha_1, \ldots, \alpha_n) \in \mathbb{F}^n$ outputs the value $f(x_1 = \alpha_1, \ldots, x_n = \alpha_n)$ (Figure 1.1).

$(\alpha_1, \ldots, \alpha_n) \in \mathbb{F}^n$
$f(\alpha_1, \ldots, \alpha_n) \in \mathbb{F}$

Figure 1.1: The black box representation

The black box model was introduced by Kaltofen and Trager [36]. They gave algorithms for computing a black box for the GCD of two polynomials given as two black boxes. Another possible representation for multivariate polynomials is the straight-line program model. In [36] Kaltofen and Trager argue that the black box representation is one the most space efficient implicit representations of multivariate polynomials and is superior to the straight-line program model in many ways.

Let $T \geq t$ be a bound on the number of non-zero terms in f and let d be a bound on the total degree of f, i.e. $d \geq \deg(f)$. Our goal is given B and possibly some information about f, such as T and/or d , we want to interpolate f with as few *probes* [1] to the black box as possible. An easy way to interpolate f is to use *Newton's* interpolation algorithm. Let $d_i = \deg_{x_i}(f) = \max(e_{1i}, e_{2i}, \ldots, e_{ti})$ be the degree of f in the i'th variable x_i and d be the degree bound such that $d_i \leq d$. Using Newton's interpolation algorithm, the number of probes to the black box is

$$N_n = \prod_{i=1}^{n} (d_i + 1) \leq (d+1)^n,$$

which is exponential in the number of variables n and is independent of t the number of terms and hence does not perform well for sparse polynomials. We will now introduce three *sparse* interpolation algorithms. The first is Zippel's algorithm which was first given by Richard Zippel in his Ph.D. thesis in 1979 [84, 83]. It was developed to solve the GCD problem. It

[1] We use the terms *probes* and *evaluations* interchangeably.

is used in several computer algebra systems including Maple, Mathematica and Magma as the default algorithm for computing multivariate GCDs over $\mathbb{Z}$. The second is an algorithm by Ben-Or and Tiwari [3] in 1988. The number of probes in both of these algorithms is sensitive to T, a bound on the number of non-zero terms in the target polynomial f. And finally the third algorithm is a hybrid of Zippel's algorithm and univariate Ben-Or/Tiwari algorithm by Kaltofen *et al.* [41, 40].

1.1.1 Zippel's Algorithm

We will describe Zippel's sparse interpolation algorithm using an example. This algorithm is *probabilistic* and similar to the dense interpolation. It interpolates the target polynomial one variable at a time.

Example 1.5. Let $p = 101$ and

$$f = 2\,x^{10}y - 14\,x^{10} + 5\,y^3x^5 - 7\,y^2 + 1 \in \mathbb{Z}_p[x, y].$$

Suppose we have a *black box* that on inputs $(x = \alpha, y = \beta)$ outputs $f(\alpha, \beta)$ mod p and for simplicity assume that we know $d_x = \deg_x(f) = 10$ and $d_y = \deg_y(f) = 3$. In Zippel's interpolation algorithm, we will first interpolate $f(x, y = \beta_1)$ mod p for some random evaluation point β_1 by evaluating $f_1^1 = f(\alpha_1, \beta_1)$ mod $p, \ldots, f_{d_x+1}^1 = f(\alpha_{d_x+1}, \beta_1)$ mod p for some random evaluation points $\alpha_1, \ldots, \alpha_{d_x+1} \in \mathbb{Z}_p^{d_x+1}$. Let $\beta_1 = 43$ (chosen at random from $\mathbb{Z}_p$). We choose $\alpha_i \in \mathbb{Z}_p$ at random and after interpolating the variable x using $f_1^1, f_2^1, \ldots, f_{11}^1$ with a dense interpolation algorithm we will get

$$f(x, y = 43) = 72\,x^{10} + 100\,x^5 + 87.$$

Note that this first step of the sparse interpolation algorithm is exactly the same as to the first step of the dense interpolation algorithm. Now we will make an important *assumption* that if we compute $f(x, y = \beta_2)$ for some evaluation point $\beta_2 \in \mathbb{Z}_p$, the resulting polynomial will have the same *terms* as $f(x, y = 43)$. This will be true if β_1 and β_2 are chosen at random from a large set of values. More precisely

$$f(x, y = \beta_2) = Ax^{10} + Bx^5 + C,$$

for some constants $A, B, C \in \mathbb{Z}$. Now lets take $\beta_2 = 93$. For the evaluation points $x =$

$45, 96, 6$ we will get the following set of equations.

$$\begin{aligned} 95A + 14B + C &= 51, \\ 36A + 6B + C &= 81, \\ A + 100B + C &= 63. \end{aligned}$$

By solving this system of linear equations we will obtain $\{A = 71, B = 66, C = 58\}$ and hence

$$f(x, y = 93) = 71\,x^{10} + 66\,x^5 + 58.$$

Now we need two more images in order to interpolate the variable y. We can compute these in the same way using the form $g_f = Ax^{10} + Bx^5 + C$ and finally after computing enough images, we will interpolate the variable y and we are done.

Remark 1.6. The number of evaluations (probes to the black box) in the Example 1.5 is $11 + 3 \times 3 = 20$. If we use dense interpolation we need at least 44 evaluations.

The main observation in Zippel's method is that after computing the first image of the polynomial for the evaluation point $x_j = \alpha_1$, we now have the very important information of what terms are present in this polynomial in variables $x_1, x_2, \ldots, x_{j-1}$ *with high probability* and we can use this information to compute other images of this polynomial evaluated at different evaluation points, say $x_j = \alpha_i$. This will be true if α_i is chosen at random from a large set of values. The following result quantifies the probability that a randomly chosen evaluation point is a root of a non-zero multivariate polynomial (See also [84]).

Lemma 1.7 (Schwartz [64])**.** Let $f \in \mathbb{F}[x_1, \ldots, x_n]$ be a non-zero polynomial and let $D = \deg(f)$ be the total degree of f. Let S be a finite subset of $\mathbb{F}$ and let $r_1, \ldots, r_n$ be random evaluation points chosen from S. We have

$$\text{Prob}(f(r_1, r_2, \ldots, r_n) = 0) \leq \frac{D}{|S|}.$$

Example 1.8. Let $f(x, y) = (x - y - 1) \times (x - y - 2) \times \cdots \times (x - y - d) \in \mathbb{Z}_p[x, y]$. We have $\deg(f) = d$. Let $S = \mathbb{Z}_p$. For each choice of $x = \alpha \in \mathbb{Z}_p$, the polynomial $f(\alpha, y)$ has exactly d roots, namely $\{\alpha - 1, \ldots, \alpha - d\}$, hence for the p^2 choices for $(x = \alpha, y = \beta) \in \mathbb{Z}_p^2$, there are $d \times p$ roots, hence the probability that $f(\alpha, \beta) = 0$ is $\frac{d}{|S|} = \frac{d}{p}$.

Remark 1.9. Let $d_i = \deg_{x_i}(f)$ and $T_{x_1,\ldots,x_j}$ be the number of terms in the polynomial f after evaluating at $x_{j+1} = \alpha_{j+1}, \ldots, x_n = \alpha_n$. The number of evaluations for Zippel's sparse interpolation method is

$$(d_1 + 1) + d_2 T_{x_1} + d_3 T_{x_1,x_2} + \cdots + d_n T_{x_1,\ldots,x_{n-1}},$$

Assuming that $d_i \leq d$ for all $1 \leq i \leq n$ and $T_{x_1,\ldots,x_j} < T$ for all $1 \leq j \leq n-1$, then the number of probes is in $O(ndT)$.

In [85] Zippel suggests one choose the evaluations points for Zippel's sparse interpolation algorithm such that the system of linear equations is a transposed Vandermonde system.

Vandermonde System of Equations

A general $n \times n$ system of linear equations can be solved with $O(n^3)$ arithmetic operations and $O(n^2)$ space using classical methods (e.g. *Gaussian Elimination*). If the system of linear equations is *structured*, one might be able to take advantage of this structure to solve the system more efficiently.

An example of a structured matrix is the *Vandermonde matrix*. A Vandermonde matrix has the following form.

$$V = \begin{pmatrix} 1 & k_1^2 & \ldots & k_1^{n-1} \\ 1 & k_2^2 & \ldots & k_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & k_n^2 & \ldots & k_n^{n-1} \end{pmatrix}$$

A *Vandermonde system of equations* ($VX = c$) is of the following form

$$\begin{aligned} X_1 + k_1 X_2 + k_1^2 X_3 + \cdots + k_1^{n-1} X_n &= c_1 \\ X_1 + k_2 X_2 + k_2^2 X_3 + \cdots + k_2^{n-1} X_n &= c_2 \\ &\vdots \\ X_1 + k_n X_2 + k_n^2 X_3 + \cdots + k_n^{n-1} X_n &= c_n. \end{aligned}$$

We use a well-known technique (See [85, 39]) for inverting a Vandermonde matrix with $O(n^2)$ operations [2] in $O(n)$ space. There are also softly linear time algorithms (e.g. see [15]). We

[2] A Vandermonde matrix is invertible if and only if $k_i \neq k_j$ for all $1 \leq i < j \leq n$.

will not consider these algorithms because they are generally not practical on the sizes of problems we are dealing with.

The *transposed Vandermonde system*, $V^T X = c$, namely

$$\begin{aligned} X_1 + X_2 + X_3 + \cdots + X_n &= c_1 \\ k_1 X_1 + k_2 X_2 + k_3 X_3 + \cdots + k_n X_n &= c_2 \\ &\vdots \\ k_1^{n-1} X_1 + k_2^{n-1} X_2 + k_3^{n-1} X_3 + \cdots + k_n^{n-1} X_n &= c_n \end{aligned}$$

can also be solved in $O(n^2)$ time and $O(n)$ space in a similar way.

Let $f \in \mathbb{F}[x_1, \ldots, x_j]$ be the target polynomial. Let $g_f = X_1 M_1 + X_2 M_2 + \cdots + X_T M_T$ be the assumed form for f where $X_1, \ldots, X_T$ are the unknown coefficients and $M_1, \ldots, M_T \in \mathbb{F}[x_1, \ldots, x_j]$ are the monomials. Suppose we choose the evaluation point $\alpha = (x_1 = \alpha_1, \ldots, x_j = \alpha_j)$ at random from $\mathbb{F}^j$. Let

$$\beta_i = \alpha^i = (x_1 = \alpha_1^i, \ldots, x_j = \alpha_j^i)$$

for $0 \le i < T$. Let $k_i = M_i(\alpha)$.

Lemma 1.10. $M_i(\beta_j) = k_i^j$.

Example 1.11. Let $g_f = X_1 x^2 y + X_2 xy + X_3 y^3 + X_4$ be the assumed form of the polynomial. Lets choose $\alpha = (x = 2, y = 3)$ hence $k_1 = 2^2 \times 3 = 12$, $k_2 = 2 \times 3 = 6$, $k_3 = 3^3 = 27$ and $k_4 = 1$. In this example $\beta_2 = \alpha^2 = (x = 4, y = 9)$. We verify that $M_1 = x^2 y$ evaluated at β_2 is k_1^2

$$M_1(\beta_2) = 4^2 \times 9 = 144 = k_1^2.$$

Lemma 1.10 implies that choosing the evaluation points $\beta_0, \ldots, \beta_{T-1}$ will result in a transposed Vandermonde system of linear equations. One of the difficulties of this method, modulo a prime p, is that α should be chosen such that $k_i = M_i(\alpha) \neq M_j(\alpha) = k_j$ for $1 \le i < j \le T$ otherwise the Vandermonde matrix will not be invertible.

1.1.2 Ben-Or/Tiwari Sparse Interpolation Algorithm

Another algorithm for sparse interpolation is the *Ben-Or/Tiwari algorithm* [3]. Unlike Zippel's algorithm, the Ben-Or/Tiwari algorithm does not interpolate the polynomial one variable at a time. The main disadvantage of this algorithm is that it needs a bound T on t, the number of terms of the polynomial which we are interpolating. Zippel's algorithm requires the bound $d \geq \deg(f)$. The Ben-Or/Tiwari algorithm computes a *linear generator* of a sequence of numbers.

Computing Linear Generators

Let $\beta_0, \beta_1, \ldots, \beta_{2T-1}, \ldots$ be a sequence of length at least $2T$ where $\beta_i \in \mathbb{K}$ for an arbitrary field $\mathbb{K}$. The univariate polynomial $\Lambda(z) = z^T - \lambda_{T-1}z^{T-1} - \cdots - \lambda_0$ is said to be a linear generator of degree T for this sequence of elements if

$$\beta_{T+i} = \lambda_{T-1}\beta_{T+i-1} + \lambda_{T-2}\beta_{T+i-2} + \cdots + \lambda_0\beta_i$$

for all $i \geq 0$. This means that each element of the sequence can be computed using the generator polynomial $\Lambda(z)$ and T previous elements in the sequence. There are various ways to find the generator $\Lambda(z)$. The naive way is to solve a system of linear equations

$$\begin{aligned}
&\beta_{T+1} = \lambda_{T-1}\beta_T + \lambda_{T-2}\beta_{T-1} + \cdots + \lambda_0\beta_1\\
&\beta_{T+2} = \lambda_{T-1}\beta_{T+1} + \lambda_{T-2}\beta_T + \cdots + \lambda_0\beta_2\\
&\beta_{T+3} = \lambda_{T-1}\beta_{T+2} + \lambda_{T-2}\beta_{T+1} + \cdots + \lambda_0\beta_3\\
&\vdots\\
&\beta_{2T} = \lambda_{T-1}\beta_{2T-1} + \lambda_{T-2}\beta_{2T-2} + \cdots + \lambda_0\beta_T
\end{aligned}$$

where $\lambda_0, \lambda_1, \ldots, \lambda_{T-1}$ are the unknowns. This costs $O(T^3)$ operations. In our implementation, we use the *Berlekamp/Massey algorithm* [55] (See [41] for a more accessible reference) which is a faster way to compute $\Lambda(z)$. The time complexity for this algorithm is $O(T^2)$. This also can be done in time softly linear in T using the Half-GCD algorithm (See [73]). But this algorithm is not practical for the size of the problems we are interested in.

The Algorithm

The Ben-Or/Tiwari algorithm evaluates the black box at points

$$\alpha_i = (2^i, 3^i, 5^i, \ldots, p_n^i)$$

for $0 \leq i < 2T$, where $p_n \in \mathbb{Z}$ is the n'th prime. Let β_i be the output of the black box on input α_i. The algorithm then computes a linear generator $\Lambda(z)$ for the sequence $\beta_0, \ldots, \beta_{2T-1}$.

Let the target polynomial be $f = A_1M_1 + A_2M_2 + \cdots + A_tM_t$ where $A_i \in \mathbb{Z}$ and M_i is a monomial in the n variables. Ben-Or and Tiwari [3] show that the roots of $\Lambda(z)$ are $m_1, \ldots, m_t$ where $m_i = M_i(p_1 = 2, p_2 = 3, \ldots, p_n)$. This means that we can find the monomials in f using the roots of $\Lambda(z)$ by doing integer divisions only. After computing the monomials, one can easily find the coefficients by solving a linear system of equations. Thus the algorithm interpolates the target polynomial with only $2T$ probes to the black box. We will illustrate this *deterministic* algorithm with an example.

Example 1.12. Let $f = 9\,x^2y - 2\,y^5 - 7\,y^3z^2 + 10$ and suppose $T = t = 4$. Suppose we choose the primes $p_1 = 2, p_2 = 3$ and $p_3 = 5$. We evaluate the black box at points $\alpha_i = (x = 2^i, y = 3^i, z = 5^i)$, for $0 \leq i \leq 7$ to obtain

$$\{\beta_0 = 10, \beta_1 = -5093, \beta_2 = -3306167, \beta_3 = -2181510377, \beta_4 = -1460132366543,$$

$$\beta_5 = -982576889432513, \beta_6 = -662507344493174807, \beta_7 = -447014567612580591257\}$$

The next step is to find a linear generator for this set of integers. Using the algorithm of Berlekamp/Massey (See [41]), we obtain

$$\Lambda(z) = z^4 - 931\,z^3 + 175971\,z^2 - 2143341\,z + 1968300.$$

We compute the roots of this polynomial [3] to obtain

$$R = \{675 = 3^3 \times 5^2, 1, 12 = 2^2 \times 3, 243 = 3^5\}.$$

Hence the monomials are

$$M_1 = y^3z^2, M_2 = 1, M_3 = x^2y, M_4 = y^5.$$

To compute the coefficients of these monomials we solve the system of linear equations where the i'th equation $(1 \leq i \leq 4)$ is

$$A_1M_1(2^i, 3^i, 5^i) + \cdots + A_4M_4(2^i, 3^i, 5^i) = \beta_i.$$

We obtain $A_1 = -7, A_2 = 10, A_3 = 9$ and $A_4 = -2$, hence the interpolated polynomial is $-7y^3z^2 + 10 + 9x^2y - 2y^5$ and we are done.

[3] One can find the roots modulo a prime p using Rabin's Las Vegas algorithm from [63] (see Chapter 8 of Geddes et. al. [18]) and then lift them using p-adic lifting to obtain the integer roots of $\Lambda(z)$ efficiently.

This algorithm is not variable by variable. Instead, it interpolates the polynomial f with $2T$ probes to the black box which can all be computed in parallel. The major disadvantage of the Ben-Or/Tiwari algorithm is that the evaluation points are large ($O(T \log n)$ bits long $-$ see [3]) and computations over $\mathbb{Q}$ encounter an expression swell which makes the algorithm very slow. This problem was addressed by Kaltofen *et al.* in [35] by running the algorithm modulo a power of a prime of sufficiently large size p^k; the modulus must be greater than $\max_j M_j(2, 3, 5, \ldots, p_n)$ where p_n is the nth prime. The Ben-Or/Tiwari algorithm will work in a finite field of characteristic p without modification when $p > p_n^d$. When interpolating a polynomial over a finite field of characteristic p, we assume that the prime p can be chosen to be a *smooth* prime. This is because for these primes, a discrete logarithm can be computed efficiently [62].

1.1.3 Hybrid of Zippel's and Ben-Or/Tiwari's Algorithms

In 2000, Kaltofen, Lee and Lobo [41] (See also [40]) introduced a hybrid of Zippel's algorithm and univariate Ben-Or/Tiwari algorithm which uses the *early termination* technique. Generally, the purpose of the early termination technique is to avoid using bounds for determining the termination point in an algorithm. As an example, in dense univariate polynomial interpolation, one can avoid using the degree bound and simply stop the algorithm when the interpolated polynomial does not change after a certain number of probes to the black box. This however does not guarantee that the algorithm always returns the correct result, hence these algorithms are probabilistic in the Monte Carlo sense. The hybrid algorithm has the same structure as Zippel's algorithm. Suppose the black box evaluates a polynomial f in n variables. And suppose we have recursively interpolated all the variables but the last one x_n, thus we have the form g for $f(x_1, x_2, \ldots, x_{n-1}, x_n = \alpha_n)$ for some $\alpha_n \in \mathbb{F}$ ($\mathbb{F}$ is a field). Let

$$g = C_1(x_n)M_1 + C_2(x_n)M_2 + \cdots + C_t(x_n)M_t,$$

where $C_i \in \mathbb{F}[x_n]$ is unknown. At this point in the algorithm, our goal is to *interpolate* the unknown univariate polynomials $C_1, \ldots, C_t$. To do this we need to have images of these polynomials. In Zippel's algorithm, we solve systems of linear equations to find images for these coefficient polynomials and then use a dense interpolation algorithm (e.g. Newton's method) to interpolate the univariate coefficients. Thus we need $d + 1$ images. Kaltofen *et* *al.* use a racing algorithm to interpolate these coefficients. They race the univariate Ben-Or/Tiwari algorithm against Newton's algorithm and the univariate interpolation stops as soon as one of these algorithms finishes by the early termination technique, i.e., after introducing a certain number of new images (usually one), the result does not change. To interpolate a univariate polynomial of degree d with t non-zero terms, the Ben-Or/Tiwari algorithm needs $2t{+}2$ points (2 extra points for early termination), while Newton's algorithm needs $d+2$ (1 extra point for early termination). Hence for sparse univariate polynomials with $2t \ll d$, the sparse algorithm of Ben-Or and Tiwari would have a better performance in terms of the number of evaluation points needed.

Example 1.13. Let $f = \left(10\,z^{10}-2\right)x^7y^8 + \left(-4\,z^3+2\right)x^3y^4 + 8\,z^4 - z^3 + 4$. Suppose that Zippel's algorithm has interpolated the variables x and y, and hence we have $f = C_1(z)x^7y^8 + C_2(z)x^3y^4 + C_3(z)$ as the assumed form for f. We need to interpolate the three univariate polynomials C_1, C_2 and C_3. To interpolate C_i's we choose a new evaluation point $(\alpha_1, \alpha_2, \alpha_3)$ and using the form $g = Ax^7y^8 + Bx^3y^4 + C$, we compute images of $f(z = \alpha_3)$. Each such image results in the image of $C_i(z)$ at $z = \alpha_3$. We race the Newton's algorithm with the algorithm by Ben-Or and Tiwari to interpolate C_i. We know that $C_1 = 10\,z^{10} - 2$, hence the Newton's interpolation algorithm needs $d + 2 = 12$ points to interpolate C_1, but the Ben-Or/Tiwari algorithm needs only $2t + 2 = 6$ points. However for $C_2 = -4\,z^3 + 2$, Newton's algorithm requires 5 points, while the sparse algorithm needs 6 points and hence the Newton's algorithm will win this race. As for $C_3 = 8\,z^5 - z^3 + 4$, both the algorithms require 6 points. Thus to interpolate f, the Ben-Or/Tiwari algorithm requires $\max(6, 6, 8) = 8$ points while Newton's algorithm needs $\max(12, 5, 6) = 12$ points hence the Ben-Or/Tiwari algorithm wins the race.

Example 1.14. Consider the polynomial $f = \sum_{i=1}^{n} x_i^d$. Zippel's algorithm requires $n(d{+}1)$ points while the racing algorithm needs only $4n$ points ($2n$ for checking when to terminate).

Remark 1.15. Let $f = \sum_{i=1}^{t_{n-1}} C_i(x_n) \times x_1^{e_{i1}} \times \cdots \times x_{n-1}^{e_{i(n-1)}}$. Suppose in the Zippel's sparse interpolation algorithm, we have interpolated the first $n-1$ variables. To interpolate the last variable, we need to interpolate the univariate polynomials $C_1, \ldots, C_{t_{n-1}}$. We need $\min(2T_i', d_i') + 2$ points to interpolate C_i using the racing algorithm with early termination strategy. Here T_i' is the number of terms in C_i and d_i' is its degree. To obtain each point, we need to solve a system of equations with t_{n-1} unknowns and hence we need to probe the black box t_{n-1} times. Hence to interpolate all C_i's, assuming that there are no optimizations implemented, i.e. no terms are pruned, we need

$$S_n = t_{n-1} \times (\max(\min(2T'_1, d'_1), \min(2T'_2, d'_2), \ldots, \min(2T'_{t_{n-1}}, d'_{t_{n-1}})) + 2)$$

probes to the black box. Thus the total number of probes to the black box to interpolate f is $N_h = \sum_{i=1}^{n} S_i$ where t_i is the number of non-zero terms in f after evaluating $x_{i+1}, \ldots, x_n$ and $S_1 = \min(2t_1, \deg_{x_1}(f)) + 2$.

1.1.4 Other Sparse Interpolation Algorithms

In 1994 Rayes, Wang and Weber in [78] looked at parallelizing Zippel's algorithm. However, because it interpolates f one variable at a time, sequentially, it's parallelism is limited. This was our motivation for looking for a new approach that we present in Chapter 2. Our approach is based on the sparse interpolation of Ben-Or and Tiwari.

In [22], Grigoriev, Karpinski and Singer present a parallel algorithm that deterministically interpolates a polynomial over GF(q) assuming that the black box can evaluate points at a extension field of cardinality q^s where $s \in O(\log_q(nt))$. The number of probes to the black box in their algorithm is polynomial in n, t and q. Their algorithm has time complexity of $\tilde{O}(n^2 t^6 + q^{2.5})$ which is not practical for large q.

In [29], Huang and Rao described how to make the Ben-Or/Tiwari approach work over finite fields GF(q) with at least $4t(t-2)d^2 + 1$ elements. Their idea is to replace the primes $2, 3, 5, \ldots, p_n$ in the Ben-Or/Tiwari algorithm by linear (hence irreducible) polynomials in GF$(q)[y]$ where y is a new variable. To do this they need a black box that evaluates the target polynomial f at the input $(p_1(y), \ldots, p_n(y))$ where $p_i(y) \in$ GF$(q)[y]$. After constructing this black box, the problem reduces to solving univariate equations and performing linear algebra computations over GF$(q)[y]$. Also to find the roots of the linear generator, one needs to do a bivariate factorization over the finite field. Their algorithm is Las Vegas and does $O(dt^2)$ probes. Although the authors discuss how to parallelize the algorithm, the factor of t^2 may limit this approach. As far as we know this algorithm has never been implemented.

In 2009, Giesbrecht, Labahn and Lee in [54] presented two new algorithms for sparse interpolation for polynomials with floating point coefficients. The first is a modification of the Ben-Or/Tiwari algorithm that uses $2T$ probes. To avoid numerical problems, it evaluates at powers of complex roots of unity of relatively prime order. In principle, this algorithm can be made to work over finite fields $GF(p)$ for applications where one can choose the prime p. One needs $p-1$ to have n relatively prime factors $q_1, q_2, \ldots, q_n$ all $> d$.

Given a primitive element α and elements $\omega_1, \omega_2, \ldots, \omega_n$ of order $q_1, q_2, \ldots, q_n$ in $GF(p)$, the exponents $(e_1, e_2, \ldots, e_n)$ of the value of a monomial $m = \omega_1^{e_1}\omega_2^{e_2}\ldots\omega_n^{e_n}$ can be obtained from the discrete logarithm; $e_i = \beta \mod p_i$ where $\beta = \log_\alpha(m)$. Finding such relatively prime numbers is not difficult. For example, for $n = 6$, $d = 30$, we find $31, 33, 35, 37, 41$ are the first five relatively prime integers greater than d. Let $q = 54316185$ be their product. We find $r = 58$ is the first even integer satisfying $r > d$, $\gcd(r, q) = 1$, and $p = rq + 1$ is prime. Now for such a prime p, discrete logarithms in $GF(p)$ can be done efficiently using the Pohlig-Hellman algorithm [62]. The running time of this algorithm is $\sum_{i=1}^{n} O(\sqrt{q_i})$. The prime p has 31.6 bits in length. In general, the prime $p > (d+1)^n$ thus the length of the prime depends linearly on the number of variables. We have not explored the feasibility of this approach.

In 2010 Kaltofen [43] suggested a similar approach. He first reduces multivariate interpolation to univariate interpolation using the Kronecker substitution $(x_1, x_2, \ldots, x_n) = (x, x^{d+1}, \ldots, x^{(d+1)^n})$ and interpolates the univariate polynomial from powers of a primitive element α using a prime $p > (d+1)^n$. If p is smooth, that is, $p - 1$ has no large prime factors, then the discrete logarithms can be computed efficiently using the Pohlig-Hellman [62] algorithm. The time complexity for computing a discrete logarithm using this algorithm is $O(\sqrt{p}) \in O(n \log(d))$. This approach has the added advantage that by choosing p smooth of the form $p = 2^k s + 1$, one can directly use the FFT in GF(p) when needed elsewhere in the algorithm.

In 2009 Garg and Schost [17] presented an interpolation algorithm to interpolate over any commutative ring S with identity which is polynomial in $\log d$. They assume the target polynomial is represented by a Straight-Line Program (SLP). Their algorithm interpolates the target polynomial in time polynomial in the size of the SLP, the number of terms in the polynomial and $\log d$ where d is the degree. For a sparse univariate polynomial $f(x)$ in $S[x]$ they evaluate f_i, the image of $f(x)$ modulo $x^{p_i} - 1$, for N primes $p_1, p_2, \ldots, p_N$, also exploiting the roots of unity. The result that they obtain requires $N > T^2 \log d$ probes, thus too many probes to be practical for large t. They prove that at least a certain number of these images have the exact number of terms as f. For each i such that $f_i(x) = \sum_{j=1}^{t} a'_j x^{e_{ij}}$ is one of these images, they compute $\chi_i(y) = \prod_{j=1}^{t}(y - e_{ij})$. Then using Chinese remaindering the algorithm computes $\chi(y) = \prod_{i=1}^{t}(y - e_i)$. The roots of this polynomial are the exponents of the monomials in f. Computing the coefficients $a_1, \ldots, a_t$ is easy. For multivariate polynomials they reduce the problem to univariate interpolation by doing Kronecker substitution. One of the advantages of this interpolation algorithm is that unlike other algorithms, there is no restriction on the size of the characteristic p. Also the number of arithmetic operations in the ring S is polynomial in $T, \log(p)$ and $\log(d)$. In [21] Giesbrecht and Roche take the idea in [17] for algebraic circuits and make it work for the black box model. The number of probes is $O(T^2 \log(d))$.

1.2 GCD Computation of Multivariate Polynomials over $\mathbb{Z}$

In this section we will discuss some methods for computing the GCD of two polynomials $f_1, f_2 \in \mathbb{Z}[x_1, \ldots, x_n]$. These include the Euclidean algorithm, the GCDHEU algorithm of Char, Geddes and Gonnet, Brown's modular algorithm, Zippel's sparse interpolation algorithm and the LINZIP algorithm of de Kleine, Monagan and Wittkopf.

1.2.1 The Euclidean Algorithm and Polynomial Remainder Sequences

The fundamental algorithm for computing polynomial GCDs is *Euclid's algorithm.* It uses the following lemma.

Lemma 1.16. Let $f_1, f_2 \in F[x]$ where $f_1, f_2 \neq 0$ and F is a field. Let q, r be the quotient and remainder of dividing f_1 by f_2. Then $\gcd(f_1, f_2) = \gcd(f_2, r)$.

Definition 1.17. Suppose we want to compute the GCD of $f_1, f_2 \in F[x]$ and $f_2 \neq 0$. Let $r_0 = f_1, r_1 = f_2$ and for i, $2 \leq i \leq k$, let $r_i = \operatorname{rem}(r_{i-1}, r_{i-2}), r_{k+1} = 0$. The sequence $r_0, r_1, \ldots, r_k$ is called the *natural Euclidean Polynomial Remainder Sequence (PRS).* The remainder r_k is an associate (scalar multiple) of $\gcd(f_1, f_2)$.

Example 1.18. Let $f_1, f_2 \in \mathbb{Z}[x]$ such that

$$f_1 = 2\,x^7 + 19\,x^6 + 3\,x^5 - 7\,x^4 + 10\,x^3 - 6\,x + 6\,x^2 - 3,$$
$$f_2 = 10\,x^6 + 17\,x^5 - 6\,x^4 - 22\,x^3 + 10\,x - 8\,x^2 + 5.$$

Note that $\mathbb{Z}$ is not a field so in order to apply the Euclidean algorithm we need to work over the field $\mathbb{Q}$. The PRS generated by applying the Euclidean algorithm to f_1 and f_2 is

$$\begin{aligned}
r_2 &= -\frac{558}{25}x^5 + \frac{169}{25}x^4 + \frac{1148}{25}x^3 + \frac{412}{25}x^2 - \frac{113}{5}x - \frac{54}{5},\\
r_3 &= \frac{1606600}{77841}x^4 + \frac{2069750}{77841}x^3 - \frac{259775}{77841}x^2 - \frac{1176850}{77841}x - \frac{13525}{2883},\\
r_4 &= -\frac{44414751303}{12905817800}x^3 + \frac{30317279157}{5162327120}x^2 - \frac{4281177159}{2581163560}x - \frac{70406172567}{25811635600},\\
r_5 &= \frac{259601712382237600}{2815811338079961}x^2 - \frac{1552655498535285200}{25342302042719649}x - \frac{1360431602127677200}{25342302042719649},\\
r_6 &= -\frac{2088055208233512819950 87}{16318476024769526530810 0}x - \frac{2088055208233512819950 87}{32636952049539053061620 0}.
\end{aligned}$$

The remainder r_7 is zero, hence the monic GCD is $\gcd(f_1, f_2) = \text{monic}(r_6) = x + \frac{1}{2}$.

There are some problems with the Euclidean algorithm as illustrated in Example 1.18. The first is the *exponential growth of the coefficients* which makes Euclid's algorithm inefficient in practice. The second problem is that we are forced to do arithmetic in $\mathbb{Q}$ instead of $\mathbb{Z}$. One way to overcome the latter problem is to use *pseudo-division*. Another problem with the Euclidean algorithm is that it is not directly applicable to multivariate polynomials over an algebraic function field. In [31] Javadi and Monagan give a primitive fraction free PRS algorithm for computing GCDs of multivariate polynomials over an algebraic function field with multiple field extensions. Unfortunately this algorithm is very slow (useless in practice) – See [31].

1.2.2 The GCDHEU Algorithm

GCDHEU is another GCD algorithm which was first introduced by Char, Geddes and Gonnet (see [8]). The name GCDHEU stands for *Heuristic GCD*. Suppose we want to find the GCD g of two univariate polynomials $f_1, f_2 \in \mathbb{Z}[x]$. Let $\gamma = \max(\gamma_1, \gamma_2)$ where γ_1 and γ_2 are the biggest coefficients in f_1 and f_2 respectively. Probably, the maximum coefficient of g is less than γ. Now we take an evaluation point $\xi \in \mathbb{Z}$ such that $\xi > 2|\gamma|$ and evaluate both of the input polynomials at this point to get $f_1(\xi), f_2(\xi) \in \mathbb{Z}$. Next we compute the *integer*

$$h = \gcd(f_1(\xi), f_2(\xi)).$$

The idea is to recover $g \in \mathbb{Z}[x]$ from the integer h. We illustrate with an example.

Example 1.19. Let

$$f_1 = 6x^4 + 21x^3 + 38x^2 + 33x + 14,$$
$$f_2 = 12x^4 - 3x^3 - 14x^2 - 39x + 28.$$

Let's take the evaluation point $\xi = 1000$. We obtain

$$f_1(\xi = 1000000) = 6000021000038000033000014,$$
$$f_2(\xi = 1000000) = 11999996999985999961000028.$$

Notice how the coefficients of f_1 and f_2 appear in the evaluations. Next we calculate the integer GCD of $f_1(\xi = 1000000)$ and $f_1(\xi = 1000000)$ (using the Euclidean algorithm) to get

$$\gcd(6000021000038000033000014, 11999996999985999961000028) = 6000012000014.$$

Notice that this corresponds to the polynomial $h = 6x^2 + 12x + 14$ with $x = 1000000$. The GCD h could have an extraneous factor, i.e. $h = \Delta g(\xi)$ where $\Delta = \gcd(\frac{f_1}{g}(\xi), \frac{f_2}{g}(\xi))$. In order to recover g from h, Δ should be removed. Here $\Delta = \mathrm{cont}_x(h) = 2$ and we have

$$g = h/2 = 3x^2 + 6x + 7.$$

Since $g \mid f_1$ and $g \mid f_2$, $g = \gcd(f_1, f_2)$ and we are done.

The algorithm can fail if Δ is too big which causes the division to fail. In this case, one can try a different evaluation point. The idea can be generalized to multivariate GCDs. It generates very large integer GCDs but can be fast if the input polynomials are dense and fast GCD computation in $\mathbb{Z}$ is available.

1.2.3 Brown's Modular GCD Algorithm

The most efficient way to solve the coefficient growth problem in the Euclidean algorithm is to use a *modular algorithm.* A modular algorithm projects the problem down to finding the answer modulo a sequence of primes and then builds up the desired answer using the Chinese remainder theorem.

Brown's algorithm (see [7]) is a modular algorithm. It uses polynomial evaluation and dense interpolation one variable at a time. Since we are computing the GCD modulo a prime p at each step, the coefficients of the polynomials can not be greater than p, therefore the coefficient growth problem will never occur. We will illustrate the algorithm using a simple example.

Example 1.20. Suppose we want to find $g = \gcd(f_1, f_2)$ where

$$f_1 = -3x^3y - x^3 - 45xy^2 - 12xy + x = (-3\,y - 1)\,x^3 + \left(-45\,y^2 - 12\,y + 1\right)x,$$
$$f_2 = x^4 + x^2 + 15x^2y + 30y - 2 = x^4 + (15\,y + 1)\,x^2 + 30\,y - 2.$$

Let the first prime p_1 be 11. Now we want to compute $g_1 = \gcd(f_1 \bmod p_1, f_2 \bmod p_1)$. We do this by first evaluating the input polynomials at some evaluation points for y, compute the corresponding univariate GCD in $\mathbb{Z}_{p_1}[x]$ using Euclidean algorithm and then *interpolate* these images to get g_1. Let's take the first evaluation point $\alpha_1 = 1$. We get

$$\begin{aligned} f_1(y = \alpha_1) \bmod p_1 &= 7x^3 + 10x, \\ f_2(y = \alpha_1) \bmod p_1 &= x^4 + 5x^2 + 6, \quad \text{and} \\ h_1 = \gcd(f_1(y = \alpha_1), f_2(y = \alpha_1)) \bmod p_1 &= x^2 + 3. \end{aligned}$$

Let's take the next evaluation point to be $\alpha_2 = 2$. We compute

$$\begin{aligned} f_1(y = \alpha_2) \bmod p_1 &= 4x^3 + 6x, \\ f_2(y = \alpha_2) \bmod p_1 &= x^4 + 9x^2 + 3, \quad \text{and} \\ h_2\,\gcd(f_1(y = \alpha_2), f_2(y = \alpha_2)) \bmod p_1 &= x^2 + 7. \end{aligned}$$

At this point we interpolate the coefficients in images h_1 and h_2 to see if we can get g_1. The output of the interpolation is

$$h = (1)x^2 + (4y + 10).$$

Since $h \mid f_1 \bmod p_1$ and $h \mid f_2 \bmod p_1$, we conclude that

$$g_1 = h = \gcd(f_1, f_2) \bmod p_1 = x^2 + 4y + 10.$$

Now we choose the next prime p_2 to be say 13. Suppose $g_2 = \gcd(f_1 \bmod p_2, f_2 \bmod p_2)$. Similar to how we computed g_1, we easily compute

$$g_2 = x^2 + 2y + 12.$$

Now applying the *Chinese Remainder theorem* to the images g_1 and g_2 we compute a candidate g' for g, the GCD we are seeking. Because in our example g is *monic* [4], if this

[4] A polynomial is *monic* if its leading coefficient in the main variable is 1.

candidate divides both of the input polynomials, then it is equal to g and we are done, otherwise we need to choose another prime p_3 and keep going until we get a candidate which divides both f_1 and f_2. Applying the Chinese remainder theorem results in

$$g' = x^2 + 15y - 1 \bmod 11 \times 13.$$

Since $g' \mid f_1$ and $g' \mid f_2$ we conclude that

$$g = g' = \gcd(f_1, f_2) = x^2 + 15y - 1,$$

and we are done.

There are some difficulties with Brown's algorithm. These include *bad primes and evaluation points*, *unlucky primes and evaluation points* and *leading coefficient reconstruction.*

Definition 1.21 (See [9])**.** Suppose $f_1, f_2 \in \mathbb{F}[x_2, \ldots, x_n][x_1]$ and $g = \gcd(f_1, f_2)$. A prime p is said to be a *bad prime* if $\deg_{x_1}(g \bmod p) < \deg_{x_1}(g)$. Similarly, an evaluation point $(\alpha_1, \ldots, \alpha_{n-1}) \in \mathbb{Z}_p^{n-1}$ is said to be a *bad* evaluation point if $\deg_{x_1}(g \bmod I) < \deg_{x_1}(g)$ where $I = \langle x_2 = \alpha_1, \ldots, x_n = \alpha_{n-1} \rangle$.

In Brown's modular GCD algorithm, bad primes and evaluation points are simply avoided in order to successfully reconstruct the GCD from its images modulo several primes.

Definition 1.22. Suppose $f_1, f_2 \in F[x_1, \ldots, x_n]$. Let $g = \gcd(f_1, f_2), a = \frac{f_1}{g}, b = \frac{f_2}{g}$. We have $\gcd(a, b) = 1$. A prime p is said to be *unlucky* if $h_p = \gcd(a \bmod p, b \bmod p) \neq 1$. Similarly an evaluation point $x_i = \alpha_j$ is *unlucky* if $h = \gcd(a(x_i = \alpha_j), b(x_i = \alpha_j)) \neq 1$.

Example 1.23. Let $f_1 = (x+y+1)(x^2-y^2+18y)$ and $f_2 = (x+y+2)(x^2-y^2+y)$. Here $g = \gcd(f_1, f_2) = 1$ but modulo the prime $p = 17$, $g_p = x^2 - y^2 + y$ so $p = 17$ is an unlucky prime. Similarly using the evaluation point $y = 0$, $h = \gcd(f_1(y=0), f_2(y=0)) = x^2$ so $y = 0$ is unlucky.

Let $g = \gcd(f_1, f_2)$. Unlucky primes and evaluation points can not be used in the modular algorithm because we need the monic images of the GCD computed modulo p to divide $g \bmod p$ to be able to recover g. Unfortunately unlucky primes and evaluation points can not be detected in advance. Brown's idea is that if during the GCD computation, one gets an image with a higher degree in the main variable compared to other images, the new image must be unlucky and hence can be discarded. The probability that a prime or evaluation point is unlucky can be made very low, if one chooses sufficiently large primes and evaluates at random points.

Brown's algorithm uses *dense interpolation.* Let $f_1, f_2 \in \mathbb{Z}[x_1, x_2, \ldots, x_n]$ and suppose we are working modulo the prime p and the main variable is x_1. In *dense* interpolation, one chooses an evaluation point $x_2 = \alpha_1 \in \mathbb{Z}_p$ and *recursively* computes h_1 the GCD of $f_1(x_2 = \alpha_1)$ and $f_2(x_2 = \alpha_1)$ modulo the prime p. Next we choose a different evaluation point $x_2 = \alpha_2 \in \mathbb{Z}_p$ and repeat the same process to compute h_2 . We do this $d + 1$ times where $d = \deg_{x_2} \gcd(f_1, f_2)$ and then use Newton's interpolation algorithm to compute a candidate for the GCD based on $h_1, h_2, \ldots, h_{d+1}$.

1.2.4 Zippel's Sparse Interpolation Algorithm

Zippel's motivational problem for developing his sparse interpolation algorithm was computing GCDs. We will illustrate this using the following example. [5]

Example 1.24. Suppose the two input polynomials are

$$f_1 = x^4 + 18x^3yz - 15x^3z^2 + 4x^2yz^2 + 14x + x^3y^2 + 18x^2y^3z - 15x^2y^2z^2 + 4xy^3z^2 + 14y^2,$$
$$f_2 = x^5 + 18x^4yz - 15x^4z^2 + 4x^3yz^2 + 14x^2 + x^3z + 18x^2yz^2 - 15x^2z^3 + 4xyz^3 + 14z.$$

Let's choose the first prime $p_1 = 11$. If we compute $g_1 = \gcd(f_1, f_2) \bmod p_1$ we get

$$g_1 = x^3 + (7yz + 7z^2)x^2 + 4yz^2x + 3.$$

Now let's take the second prime p_2 to be 13. Assuming that g_1 is of correct form, we have

$$g_2 = \gcd(f_1, f_2) \bmod p_2 = Ax^3 + (Byz + Cz^2)x^2 + Dyz^2x + E,$$

for some constants A, B, C, D and E. To find these constants we compute some *univariate* GCDs in order to obtain some linear equations. Take the first evaluation point $\alpha_1 = (y = 1, z = 1)$. We have

$$h_1 = \gcd(f_1(\alpha_1), f_2(\alpha_1)) \bmod p_2 = x^3 + 3x^2 + 4x + 3.$$

[5] For simplicity we will choose an example with a *monic* GCD. Later we will see that if the GCD is not monic in the main variable, we can not use Zippel's algorithm directly.

If we plug in the first evaluation point α_1 into our assumed form for the GCD we get

$$Ax^3 + (B + C)x^2 + Dx + E = x^3 + 3x^2 + 4x + 3.$$

From this we get the following linear equations modulo 13

$$\{A = 1, B + C = 3, D = 4, E = 1\}.$$

We still don't know the exact values of B and C so we need another image. Take the second evaluation point $\alpha_2 = (y = 2, z = 3)$, After evaluating f_1 and f_2 at the new evaluation point and computing the univariate image we obtain

$$h_2 = \gcd(f_1(\alpha_2), f_2(\alpha_2)) \bmod p_2 = x^3 + 12x^2 + 7x + 1.$$

Again we plug in the second evaluation point α_2 into the assumed form for the GCD to get

$$Ax^3 + (6B + 9C)x^2 + Dx + E = x^3 + 12x^2 + 7x + 1.$$

So we have

$$6B + 9C = 12 \bmod 13.$$

From this equation, and the equation $B + C = 3$ we find that $B = 5 \bmod 13$ and $C = 11 \bmod 13$. This means that

$$g_2 = Ax^3 + (Byz + Cz^2)x^2 + Dyz^2x + E = x^3 + (5yz + 11z^2)x^2 + 4yz^2x + 1.$$

Since $g_2 \mid f_1 \bmod 13$ and $g_2 \mid f_2 \bmod 13$, we conclude that $g_2 = \gcd(f_1, f_2) \bmod p_2$. We can find other images of the GCD using the same method as above. If we had used $p_1 = 11$ then the method would fail because the term $11z^2x^2$ would vanish.

1.2.5 LINZIP Algorithm and the Normalization Problem

If the GCD of two polynomials is not monic in the main variable x, the sparse modular GCD algorithm of Zippel can not be applied directly as one is unable to scale univariate images of the GCD in x consistently. This is called the *normalization problem.* To solve this, Zippel in his implementation of the GCD algorithm in Macsyma, normalized by $\Delta = \gcd(\mathrm{lc}_{x_1}(f_1), \mathrm{lc}_{x_1}(f_2)) \in \mathbb{Z}[x_2, \ldots, x_n]$. The algorithm scales the univariate images of the GCD by $\Delta(\alpha_2, \ldots, \alpha_n)$. This results in interpolating $\frac{\Delta}{\mathrm{lc}_{x_1}(g)} g$ which could be a much bigger polynomial than g.

The problem is that the univariate images of the GCD computed using the Euclidean algorithm are unique up to a scalar multiple (we choose them to be monic). So in order to be able to use these images to solve the system of linear equations we must scale the univariate image by the image of the leading coefficient [6] evaluated at the same evaluation point.

An ingenious solution to the normalization problem is given by Wang [78, 77]. Wang determines the leading coefficient by factoring the leading coefficient of one of the input polynomials and heuristically determining which part belongs to the GCD and which part belongs to the cofactor. We will discuss this later in Section 1.3.1.

Another solution to the normalization problem is given by Monagan *et al.* in LINZIP algorithm [9]. The main idea here is to assume that the leading coefficient of the i'th univariate image is an unknown m_i (a scaling multiple). So we multiply the i'th image by m_i. This means after each univariate GCD computation we will add one new unknown and *hopefully* t new equations to the system of linear equations where t is the number of terms in the univariate GCD. This means that the number of univariate images needed is

$$\max(\frac{N}{t-1}, n_{max}),$$

where N is the number of unknowns in the assumed form of the GCD [7] and n_{max} is the maximum number of terms in any coefficient of the GCD in the main variable x_1.

The following is an example from [9].

Example 1.25. Suppose we want to compute the GCD $g = (3y^2 - 90)x^3 + 12y + 100$. And using the first image modulo the first prime, we know that the assumed form for the GCD is $g_f = Ax^3y^2 + Bx^3 + Cy + D$. We choose the next prime $p = 17$ and the evaluation points $y = 1, 2, 3$ to obtain the following system of linear equations.

$$\begin{aligned}
(A + B)x^3 + C + D &= m_1(x^3 + 12), \\
(4A + B)x^3 + 2C + D &= m_2(x^3 + 8), \\
(9A + B)x^3 + 3C + D &= m_3(x^3),
\end{aligned}$$

[6] In fact we could scale based on the image of *any* coefficient of the GCD and not only the leading coefficient.

[7] More precisely $N = (\sum_{i=1}^{t} n_i) - 1$, where n_i is the number of terms in the i'th coefficient.

where m_1, m_2 and m_3 are the scaling factors. Note that since the GCD is only unique up to a scalar, we can always set $m_1 = 1$. After solving this linear system we will get $A = 7, B = 11, C = 11, D = 1, m_2 = 5, m_3 = 6$.

This solution to the normalization problem does not require any factorization (which could be expensive). One of the problems with this solution is that the new system of linear equations (compared to the single scaling case in Zippel's method) is bigger and the systems of linear equations are no longer *independent* (because of the introduction of scaling factors). Monagan *et al.* [9] claim that the cost of solving the linear system is the same as the single scaling case but one loses the ability to solve the systems in *parallel* because of their dependency.

Univariate Rational Function Reconstruction

Suppose we are using a modular algorithm to compute the GCD of $f_1, f_2 \in \mathbb{F}[x_1, \ldots, x_n]$. Let $g = C_1(x_n)M_1 + C_2(x_n)M_2 + \cdots + C_t(x_n)M_t \in \mathbb{F}[x_n][x_1, \ldots, x_{n-1}]$ be the GCD and $M_i \in \mathbb{F}[x_1, \ldots, x_{n-1}]$ be a monomial in g. When we use a modular algorithm, we first compute the image of the GCD $g_i \in \mathbb{F}[x_1, \ldots, x_{n-1}]$ for a series of evaluation points $x_n = \alpha_1, x_n = \alpha_2, \ldots, x_n = \alpha_{d+1} \in \mathbb{F}$. We compute $g' = M_1 + \frac{C_2(x_n)}{C_1(x_n)}M_2 + \cdots + \frac{C_t(x_n)}{C_1(x_n)}M_t \in F(x_n)[x_1, \ldots, x_{n-1}]$ and then clear the denominator. The problem is that we need to interpolate the rational functions in the coefficients of g'. We do this in two steps. We first use the Chinese remaindering algorithm. The output of the Chinese remaindering algorithm is $g' = M_1 + C_2'(x_n)M_2 + \cdots + C_t'(x_n)M_t$ where $C_i'(x_n) \equiv \frac{C_i(x_n)}{C_1(x_n)} \bmod (x_n - \alpha_1) \times \cdots \times (x_n - \alpha_{d+1})$. To compute g, we need to recover the rational function $\frac{C_i(x_n)}{C_1(x_n)}$ from $C_i'(x_n)$. To do this, we use the extended Euclidean algorithm.

Definition 1.26. Let F be a field and let $m, u \in F[x]$ where $0 \le \deg(u) < \deg(m)$. The problem of *Rational Function Reconstruction* is given m and u, find a rational function $n/d \in F(x)$ such that

$$n/d \equiv u \bmod m,$$

satisfying $\gcd(m, d) = \gcd(n, d) = 1$.

Recall that on inputs m and u, the *extended Euclidean algorithm* computes a sequence of triples s_i, t_i, r_i satisfying

$$s_i m + t_i u = r_i.$$

Hence we have

$$t_i u \equiv r_i \mod m.$$

Thus for i satisfying $\gcd(t_i, m) = 1$, the rationals $\frac{r_i}{t_i}$ satisfy $\frac{r_i}{t_i} \equiv u \bmod m$ and hence are possible solutions for our problem.

Example 1.27. Let $F = \mathbb{Z}_7, u = x^2 + 5x + 6$ and $m = (x-1)(x-2)(x-3)$. Using the Extended Euclidean Algorithm we get the following set of solutions

$$S = \left\{ \frac{x^2+5x+6}{1}, \frac{1}{x^2+2}, \frac{3x+3}{x+3} \right\}.$$

The solution to the rational function reconstruction is not always unique. We can force the uniqueness by choosing degree bounds $\deg(n) \leq N$ and $\deg(d) \leq D$ satisfying $N + D < \deg(m)$. As an example, if we had bounds $N = 1$ and $D = 1$ in Example 1.27, the unique answer is

$$\frac{n}{d} = \frac{3x+3}{x+3}.$$

Maximal Quotient Rational Reconstruction

Let $n, d \in \mathrm{GF}(q)[x]\backslash\{0\}$ be relatively prime and let $m \in \mathrm{GF}(q)[x]\backslash\{0\}$ be relatively prime to d. Let q_{i+1} be the quotient of dividing r_{i-1} by r_i in the extended Euclidean algorithm. The following lemma is from [57].

Lemma 1.28. $\deg r_i + \deg t_i + \deg q_{i+1} = \deg m$.

Lemma 1.28 suggests that if q_{i+1} has large degree , i.e. $\deg q_{i+1} \geq T$ for some T, then we should output the rational $\frac{r_i}{t_i}$. The maximal quotient rational reconstruction (MQRR) algorithm will output $\frac{r_i}{t_i}$ for q_{i+1} the quotient of maximal degree provided $\deg q_{i+1} \geq T$. Let $d_m = \deg(m)$. For $T = \frac{d_m}{2}$ the MQRR algorithm will output the correct answer with probability 1. In [57] Monagan conjectures that the probability of MQRR making an error is $O(\frac{d_m}{q^{T-1}})$. In Section 3.2 we will prove the probability that MQRR fails is at most $\frac{d_m^T}{q^{T-1}}$.

1.3 Polynomial Factorization

Let $f \in \mathbb{F}[x_1, \ldots, x_n]$ be a multivariate polynomial where $\mathbb{F}$ is a field. Our general problem is given f, find monic *irreducible*[8] polynomials $f_1, \ldots, f_r \in \mathbb{F}[x_1, \ldots, x_n]$ such that

[8] A polynomial $g \in \mathbb{F}[x_1, \ldots, x_n]$ is irreducible if and only if $g \notin \mathbb{F}$ and for any $h_1, h_2 \in \mathbb{F}$ with $g = h_1 h_2$ we have $h_1 \in \mathbb{F}$ or $h_2 \in \mathbb{F}$.

$f = uf_1 f_2 \dots f_r$ where $u \in \mathbb{F}$. We assume f is square-free[9]. A factorization is unique up to the order of the factors and multiplication by units in $\mathbb{F}$. For simplicity and without loss of generality we assume $r = 2$, i.e. the input polynomial factors into two irreducible polynomials $f = uf_1 f_2$.

In the next section we will discuss factorization over the field of rationals.

1.3.1 The EEZ Algorithm and Hensel Lifting

The EEZ algorithm presented by Wang [77] is one the most efficient algorithms for factoring multivariate polynomials over integers. It uses *multivariate Hensel lifting.*

We will describe the EEZ algorithm briefly. Suppose we want to factor the polynomial $f \in \mathbb{Q}[x_1, \dots, x_n]$. By clearing the denominator in f one can reduce this problem to factoring over $\mathbb{Z}$. Without loss of generality, assume f is primitive and factors into two irreducible factors $f = f_1 f_2$. The algorithm first does a univariate factorization in $\mathbb{Z}_{p^l}[x_1]$ where l is an integer such that $\frac{p^l}{2}$ bounds the magnitudes of all the coefficients appearing in f, f_1 and f_2. The algorithm does arithmetic in $\mathbb{Z}_{p^l}$ (instead of $\mathbb{Z}$) to avoid computing with fractions. Let $\alpha = (x_2 = \alpha_2, \dots, x_n = \alpha_n)$ be the evaluation point. After factoring the univariate polynomial $f(\alpha)$ we will have

$$f \equiv f_{1,1} \times f_{2,1} \bmod \langle x_2 - \alpha_2, \dots, x_n - \alpha_n \rangle .$$

The algorithm then heuristically computes the leading coefficients of f_1 and f_2 by factoring

$$l = \mathrm{lc}_{x_1}(f) \in \mathbb{Z}[x_2, \dots, x_n]$$

recursively (details are given in Section 1.3.1). Let $l_1 = \mathrm{lc}_{x_1}(f_1)$ and $l_2 = \mathrm{lc}_{x_1}(f_2)$ where $l_1, l_2 \in \mathbb{Z}_{p^l}[x_2, \dots, x_n]$. Suppose $\tilde{f_1}$ and $\tilde{f_2}$ are obtained from $f_{1,1}$ and $f_{2,1}$ by replacing their leading coefficients in x_1 by l_1 and l_2 respectively. Note that $\tilde{f_1}, \tilde{f_2} \in \mathbb{Z}_{p^l}[x_1, \dots, x_n]$. Set $f_{1,1} := \tilde{f_1}$ and $f_{2,1} := \tilde{f_2}$. The algorithm then *lifts* the variables one by one. For the j'th variable x_j suppose we have lifted variables $x_2, \dots, x_{j-1}$ to get $f_{1,j-1}, f_{2,j-1} \in \mathbb{Z}_{p^l}[x_1, \dots, x_n]$ such that

$$f \equiv f_{1,j-1} \times f_{2,j-1} \bmod \left\langle x_j - \alpha_j, \dots, x_n - \alpha_n, p^l \right\rangle .$$

[9] A polynomial $f \in \mathbb{F}[x_1, \dots, x_n]$ is square free, if there is no irreducible polynomial g such that $g^2 \mid f$ over $\mathbb{F}$.

Let $f_{1,j}^1 = f_{1,j-1}$ and $f_{2,j}^1 = f_{2,j-1}$. Now to lift x_j, for $1 \le k \le \deg_{x_j} f(x_{j+1} = \alpha_{j+1}, \ldots, x_n = \alpha_n)$, we will compute $f_{1,j}^k$ and $f_{2,j}^k$ such that

$$f(x_{j+1} = \alpha_{j+1}, \ldots, x_n = \alpha_n) \equiv f_{1,j}^k \times f_{2,j}^k \bmod \left\langle (x_j - \alpha_j)^k, p^l \right\rangle.$$

This is done by solving a multivariate Diophantine equation. For details, see [78, 77, 18]. Let

$$e_j^k = f(x_{j+1} = \alpha_{j+1}, \ldots, x_n = \alpha_n) - f_{1,j}^k \times f_{2,j}^k.$$

If $e_j^k = 0$, we will set $f_{1,j} = f_{1,j}^k$ and $f_{2,j} = f_{2,j}^k$ and move on to the next variable x_{j+1}.

We will not go into the details of the problems that may arise for the EEZ algorithm. We refer the reader to [78, 77] for the extensive discussion of these problems and how to overcome them.

In the following section we will discuss the details of how to determine the leading coefficients of f_1 and f_2.

Determining the leading coefficient

In [78, 77] Wang presents a method for determining the leading coefficients of f_1 and f_2 by first factoring $l = \mathrm{lc}_{x_1}(f) \in \mathbb{Z}[x_2, \ldots, x_n]$ recursively using the EEZ algorithm in Section 1.3.1. Let

$$l = \Omega \times U_1^{e_1} \times U_2^{e_2} \times \cdots \times U_k^{e_k},$$

where Ω is an integer and the U_is are distinct irreducible polynomials of positive degree. To determine the leading coefficients, the evaluation point $\alpha = (x_2 = \alpha_2, \ldots, x_n = \alpha_n)$ must satisfy the following restriction: The integer $\tilde{U}_i = U_i(\alpha)$ has at least one prime divisor p_i which does not divide Ω or $\tilde{U}_j$ for all $j < i$ or $\delta = \mathrm{cont}(f_0)$ where $f_0 = f(\alpha)$.

Let $\bar{f} = f_0/\delta = \bar{f}_{1,1} \times \bar{f}_{2,2}$. We want to compute a polynomial $\Delta(x_2, \ldots, x_n)$ which is a scalar multiple of l_1 the leading coefficient of f_1.

We illustrate this with the following example.

Example 1.29. Let $f = f_1 \times f_2 \in \mathbb{Z}[x, y, z]$ where

$$f_1 = (y^2 - z^2)x^2 + y - z^2,$$
$$f_2 = zx^2 + 2\,y + 3\,zx.$$

We have $l = \mathrm{lc}_x(f) = zy^2 - z^3 \in \mathbb{Z}[y, z]$. The first step is to factor l to obtain

$$l = z(y + z)(y - z).$$

Hence $\Omega = 1, U_1 = z, U_2 = y + z, U_3 = y - z$ and $e_1 = e_2 = e_3 = 1$. The evaluation point $\alpha = (y = 5, z = -12)$ satisfies the required condition because the integers in the set

$$\{\tilde{U}_1 = -12, \tilde{U}_2 = -7, \tilde{U}_3 = 17\}$$

have distinct prime divisors that do not divide Ω. Now we factor the univariate polynomial $\bar{f} = f(\alpha)$ to obtain

$$\bar{f} = \bar{f}_1 \times \bar{f}_2 = (119x^2 + 139)(12x^2 + 36x - 10).$$

Now the observation is that $\tilde{U}_2 \mid \mathrm{lc}_x(\bar{f}_1) = 119$ but $\tilde{U}_2 \nmid \mathrm{lc}_x(\bar{f}_2) = 12$ hence we conclude that $U_2 = y + z \mid l_1 = \mathrm{lc}_x(f_1)$. Similarly $\tilde{U}_3 \mid 119$ but $\tilde{U}_1 \nmid 119$ thus we determine that $l_1 = z^2 - y^2$ and $l_2 = \frac{l}{l_1} = z$.

1.3.2 Gao's Algorithm

In [16], Gao presents an algorithm for factoring multivariate polynomials with coefficients from a field $\mathbb{F}$ of characteristic zero. The first step is to reduce the multivariate factorization of $\mathbb{F}[x_1, \ldots, x_n]$ to bivariate factorization in $\mathbb{F}[x, y]$ by substituting $x_i = a_i x + b_i y + c_i$ for some random a_i, b_i and c_i. Then using a simple partial differential equation, a system of linear equations is obtained. The irreducible factorization of the bivariate polynomial can be obtained by solving this linear system. Let d be the total degree of the input polynomial. The degree of the bivariate polynomial in x and y is d. The size of the linear system obtained is $O(d^2)$ which may be very big. Using Gaussian elimination, solving the linear systems costs $O(d^6)$. A careful implementation of this algorithm is needed to investigate the feasibility of using this algorithm in practice.

In the next section we will discuss factorization over algebraic fields.

1.3.3 Polynomial Factorization over Algebraic Fields

The problem of factoring polynomials over algebraic fields has been of interest for a long time. In 1882, Kronecker [44] suggested the use of *norms* for factorization. A similar idea was later presented by van der Waerden in [68]. In 1976, Trager [67] improved this idea and presented an algorithm for factoring polynomials over an algebraic number field L with one field extension. His method can easily be generalized to algebraic function fields with multiple extensions. Trager's algorithm is currently used in several computer algebra systems such as Maple 14 and Magma 2.16-13.

1.3.4 Trager's Algorithm

The basic idea in Trager's algorithm is to map the polynomial f from the algebraic field to a polynomial h over the rationals such that each factor of f can be computed from a factor of h using a GCD computation over L.

Definition 1.30 ([67])**.** Let $L = \mathbb{Q}(\alpha)$ be an algebraic number field. Let $m(z)$ be the minimal polynomial for α. We have $m(\alpha) = 0$. Let $\alpha_2, \alpha_3, \ldots, \alpha_d$ be the remaining distinct roots of $m(z)$. Any $\beta \in L$, can uniquely be represented as a polynomial P in α with degree less that $d = \deg_z(m)$. The conjugates of β are $P(\alpha_2), P(\alpha_3), \ldots$. The product of β and its conjugates is called the *norm* of β. We have $\text{norm}(\beta) \in \mathbb{Q}$.

Example 1.31. Let $L = \mathbb{Q}(\sqrt{2})$. We have $\alpha_2 = -\sqrt{2}$ is a conjugate of $\alpha = \sqrt{2}$. Let $\beta = 2\sqrt{2} + 7$. Here β has one conjugate $\beta_2 = -2\sqrt{2} + 7$. We have $\text{norm}(\beta) = \beta \times \beta_2 = 41$.

Let $L = F[z_1, \ldots, z_r]/\langle m_1, \ldots, m_r\rangle$. For a polynomial $f \in L[x_1, \ldots, x_n]$, the norm$(f)$, in terms of resultants, is defined as follows (e.g. see [73]). Let

$$h_r = \text{res}_{z_r}(f, m_r),$$
$$h_i = \text{res}_{z_i}(h_{i+1}, m_i), 1 \le i < r.$$

Define $\text{norm}(f) = h_1$. Note that $h \in \mathbb{Z}[t_1, \ldots, t_k, x_1, \ldots, x_n]$ does not have $z_1, \ldots, z_r$.

Trager [67] shows that if $f \in L[x_1, \ldots, x_n]$ is square-free and irreducible over L, then norm(f) is a power of an irreducible polynomial over $\mathbb{Q}$. Also he shows that if $\text{norm}(f) = f_1 \times f_2 \times \cdots \times f_j$ with $\gcd(f_i, f_l) = 1$, then $g_i = \gcd(f, f_i) \in L[x_1, \ldots, x_n]$ is monic and irreducible and $f = a \prod_{i=1}^{j} g_i$ for some scalar $a \in L$.

Example 1.32. Let $L = \mathbb{Q}(\alpha)$ where $\alpha = \sqrt{2}$ and $f = (-4750\alpha + 3990)x^4 - 3800x^3 + (3342\alpha - 2872)x - 40\alpha + 2640$. We have

$$\text{norm}(f) = -29204900x^8 - 30324000x^7 + 14440000x^6 + 40579440x^5 +$$
$$42134400x^4 - 20064000x^3 - 14089544x^2 - 14629440x + 6966400.$$

We factor norm of f over integers to obtain

$$\mathrm{norm}(f) = -4f_1f_2 = -4\,\left(809\,x^2 + 840\,x - 400\right)\left(9025\,x^6 - 12540\,x^3 + 4354\right).$$

We have $g_1 = \gcd(f, f_1) = 809x + 500\alpha + 420$ is a factor of f. The other factor is $g_2 = \gcd(f, f_2) = 95x^3 + \alpha - 66$.

Trager's algorithm does some GCD computations over algebraic fields, i.e. $g_i = \gcd(f, f_i)$. Hence having a good GCD algorithm improves his algorithm significantly. In [56] we show that using the SparseModGcd algorithm, instead of ModGcd, results in a considerable improvement.

Trager's algorithm in [67] reduces factorization over $\mathbb{Q}(\alpha)[x]$ to $\mathbb{Z}[x]$. If f has degree l in x and α has degree d over $\mathbb{Q}$, then norm$f$ has degree ld. Thus a polynomial time algorithm for factoring over $\mathbb{Q}(\alpha)[x]$ is obtained. However if f is multivariate, the size of norm(f) can be much larger (i.e. $O(n^d)$).

Example 1.33. Consider the following polynomial from Kotsireas [19].

$$f = \frac{19}{2}c_4^2 - \sqrt{11}\sqrt{5}\sqrt{2}c_5c_4 - 2\sqrt{5}c_1c_2 - 6\sqrt{2}c_3c_4 + \frac{3}{2}c_0^2 + \frac{23}{2}c_5^2 +$$

$$\frac{7}{2}c_1^2 - \sqrt{7}\sqrt{3}\sqrt{2}c_3c_2 + \frac{11}{2}c_2^2 - \sqrt{3}\sqrt{2}c_0c_1 + \frac{15}{2}c_3^2 - \frac{10681741}{1985}.$$

Here $L = \mathbb{Q}(\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{7}, \sqrt{11})$ is a number field and $f \in L[c_0, \ldots, c_5]$. The norm of f is degree 64 in $c_0, c_1, c_2, c_3, c_4, c_5$ and has about 3 million terms and the integers in the rational coefficients have over 200 digits so it is not easy to compute norm(f) let alone factor it. But we can easily discover that f is irreducible over L by evaluating the variables $c_0, \ldots, c_4$ at small integers and then using Trager's algorithm to factor norm(f), a polynomial of degree 64 in c_5 over $\mathbb{Q}$.

1.3.5 Other Algorithms

In 1985, Landau [47] proved that for an algebraic number field L of degree m, a univariate polynomial of degree n with coefficients in L can be factorized using Trager's algorithm in time polynomial in n and m. This is true if a polynomial time algorithm is used to factor the norm over rationals (e.g. [72, 69]).

Also in 1976 [75, 76] P. S. Wang presented an algorithm for factoring multivariate polynomials over an algebraic number field. He assumed monic examples. This algorithm reduced the factorization to univariate by evaluating all the variables but one at random evaluation points. The algorithm then recovers the true factors by using Hensel lifting. Since the polynomials have algebraic numbers as coefficients, one can solve the leading coefficient problem by multiplying the original polynomial by the inverse of its leading coefficient (in all the variables) to make it monic. To factor the univariate polynomial, one can use Trager's algorithm.

Another algorithm in 1976 is by Weinberger and Rothschild [81]. Their algorithm is for factoring a univariate polynomial over an algebraic number field. By using the Chinese remainder theorem in a certain way, Weinberger and Rothschild generalize the Berlekamp-Zassenhaus algorithm so that the coefficients of the polynomial to be factored may be in $\mathbb{Q}(\alpha)$ where α is the algebraic extension. In practice, their algorithm can be very slow. In [1], Abbott gives an example where the polynomial and the extension field are both of degree n and this algorithm needs to do more than n^{n^2} Chinese remainder operations.

In 1980's Lenstra presented several algorithms (See [51, 49, 52]) for factoring polynomials over a number field which are all generalizations of the polynomial time algorithm for factoring over rationals given in [50] by Lenstra *et al.* These algorithms use lattice base reduction techniques. In [52] the author mentions that although these algorithms are polynomial-time, they are not useful for practical purposes. This is because the basis reduction algorithm needs to be applied to huge dimensional lattices with large entries. Recently there have been significant improvements in factoring univariate polynomials over the integers using a refined lattice reduction technique [24] as well as improvements in the lattice reduction itself. It would be interesting to re-examine [47, 52] in light of these developments.

One of the first algorithms for factoring non-monic polynomials over an algebraic function field which uses Hensel lifting is due to Abbott [1]. To determine the leading coefficients of the factors, his algorithm uses a method given in [38] by Kaltofen. Suppose we can somehow lift the univariate factors in $L[x_1]$ to bivariate factors in $L[x_1, x_2]$. The leading coefficient of the factors in x_2 are the complete univariate factorization of the leading coefficient of the original polynomial which has one less variable and hence can be computed recursively. The problem with his algorithm is that the bound on the size of the numerical coefficients are bad and hence one needs to do the Hensel lifting modulo p^k for a large k which can make the algorithm to be very slow.

In [74] D. Wang demonstrates an algorithm for factoring a univariate polynomial over an algebraic function field which is done by computing characteristic sets. To overcome the leading coefficient problem, the algorithm finds a *normalization* of the original polynomial which only has the parameters, and not the algebraic variables, in the leading coefficient. This is done by multiplying the polynomial by a factor of the norm of the leading coefficient. Later in 2000 Zhi [82] generalized this for multivariate polynomials by the use of Hensel lifting.

1.4 Outline of Thesis

In Chapter 2 we will present a new algorithm for parallel interpolation of a sparse multivariate polynomial represented with a black box with coefficients over a finite field. Our new algorithm is a generalization of the Ben-Or and Tiwari algorithm. For sparse polynomials, it does about a factor of $O(d)$ less probes to the black box compared to Zippel's algorithm. Both our new algorithm and the racing algorithm by Lee and Kaltofen do $O(nT)$ probes to the black box for sparse polynomials, but unlike Zippel's algorithm and the racing algorithm, our new algorithm is highly parallelized. We have done a parallel implementation in Cilk [66]. The benchmarks (See Section 2.4) show a linear speed-up which looks very promising. This work was published in the proceedings of the PASCO 2010 conference [33].

In Chapter 3 we present three new contributions in computing GCDs of polynomials over algebraic fields. The first is an in-place implementation of the Euclidean algorithm over an algebraic number field. Computing the univariate images of the GCD is the bottleneck of the SparseModGcd algorithm [31, 30] for sparse polynomials. The main idea is to eliminate all calls to the storage manager by pre-allocating one large piece of working storage and re-using parts of it in a computation. This resulted in a considerable improvement (See Chapter 3 for benchmarks). This was published in the proceedings of ASCM '09 conference [32]. The implementation of this algorithm was added to Maple 14. The second contribution is we prove that one can eliminate the trial divisions in the positive characteristic in the modular algorithms. When we compute images of the GCD modulo a prime p, we use the maximal quotient rational reconstruction (MQRR) algorithm to recover the coefficients of the GCD. The idea is to give an upper bound on the probability that the MQRR returns a wrong result. Using this we prove that if we eliminate the trial divisions in the positive characteristic, the SparseModGcd algorithm will terminate and return the correct result. This yields to significant improvement especially when the GCD is dense. Our third contribution is a new solution for the normalization problem. The SparseModGcd algorithm used to use the multiple scaling factor method from the LINZIP algorithm [9]. Our contribution here is twofold. First, we show that there is an error in this method. Second, when we use this idea, we lose the ability to have Vandermonde systems of equations in Zippel's interpolation algorithm. We give a new method to solve the normalization problem and prove that it works correctly. Using this new method, we get Vandermonde systems of equations.

In Chapter 4 we will give a new efficient algorithm for factoring multivariate polynomials over algebraic number and function fields. Our new algorithm is a generalization of Wang's EEZ algorithm. It does polynomial evaluation and interpolation using Hensel lifting. To determine the leading coefficients of the univariate factors, we use the norms of the factors of the leading coefficient in the main variable evaluated at some evaluation point and the integer denominators of the monic univariate factors. There are no good bounds on the size of the integer coefficients in the factors of the polynomials over algebraic fields. To avoid using bad bounds, we do Hensel lifting modulo a machine prime and then lift the integer coefficients using a new method called sparse p-adic lifting. This work was published in the proceedings of ISSAC '09 [34]. This algorithm will appear in Maple 16.

Chapter 2

Parallel Sparse Interpolation

Let p be a prime and $f \in \mathbb{Z}_p[x_1, \ldots, x_n]$ be a multivariate polynomial with $t > 0$ non-zero terms which is represented by a *black box* $\mathbf{B} : \mathbb{Z}_p^n \to \mathbb{Z}_p$. On input of $(\alpha_1, \ldots, \alpha_n) \in \mathbb{Z}_p^n$, the black box evaluates and outputs $f(x_1 = \alpha_1, \ldots, x_n = \alpha_n)$. Given also a degree bound $d \geq \deg f$ on the total degree of f, our goal is to interpolate the polynomial f with minimum number of evaluations (probes to the black box). Newton interpolation needs $O(n^{d+1})$ points to interpolate f which is exponential in d. For sparse f, that is, $t \ll n^{d+1}$, we seek algorithms whose computational complexity is polynomial in t, n, d and $\log p$.

Sparse interpolation plays a key role in several algorithms in computer algebra such as algorithms for polynomial GCD computation [84, 30, 9] and solving systems of polynomial equations involving parameters over $\mathbb{Q}$. In these applications one solves the problems modulo a prime p where p is usually chosen to be a machine prime, typically 31 or 63 bits.

Our approach for sparse interpolation over $\mathbb{Z}_p$ is to use evaluation points of the form $(\alpha_1^i, \ldots, \alpha_n^i) \in \mathbb{Z}_p^n$ and modify the Ben-Or/Tiwari algorithm to do extra probes to determine the degrees of the variables in each monomial in f. We do $O(nt)$ probes in order to recover the monomials from their images. The main advantage of our approach is the increased parallelism.

This chapter is organized as follows. In Section 2.1 we present an example showing the main flow and the key features of our algorithm. We then identify possible problems that can occur and how the new algorithm deals with them in Section 2.2. In Section 2.3 we present our new algorithm and analyze its time complexity. Finally, in Section 2.4 we compare the C implementations of our algorithm and Zippel's algorithm with Kaltofen *et al.* racing algorithm [40] on various sets of polynomials.

2.1 The Idea and an Example

Let $f = \sum_{i=1}^{t} C_i M_i \in \mathbb{Z}_p[x_1, \ldots, x_n]$ be the polynomial represented with the black box $\mathbf{B} : \mathbb{Z}_p^n \to \mathbb{Z}_p$ with $C_i \in \mathbb{Z}_p \backslash \{0\}$. Here t is the number of non-zero terms in f. $M_i = x_1^{e_{i1}} \times x_2^{e_{i2}} \times \cdots \times x_n^{e_{in}}$ is the i'th monomial in f where $M_i \neq M_j$ for $i \neq j$. Let $T \geq t$ be a bound on the number of non-zero terms and let $d \geq \deg f$ be a bound on the degree of f so that $d \geq \sum_{j=1}^{n} e_{ij}$ for all $1 \leq i \leq t$. We demonstrate our algorithm on the following example. Here we use x, y and z for variables instead of x_1, x_2 and x_3.

Example 2.1. Let $f = 91yz^2 + 94x^2yz + 61x^2y^2z + 42z^5 + 1$ and $p = 101$. Here $t = 5$ terms and $n = 3$ variables. We suppose we are given a black box that computes f and we want to interpolate f. We will use $T = 5$ and $d = 5$ for the term and degree bounds. The first step is to pick n non-zero elements $\alpha_1, \alpha_2, \ldots, \alpha_n$ from $\mathbb{Z}_p$ at random. We evaluate the black box at the points

$$(\alpha_1^i, \alpha_2^i, \ldots, \alpha_n^i) \text{ for } 0 \leq i < 2T.$$

Thus we make $2T$ probes to the black box. Let $V = (v_0, v_1, \ldots, v_{2T-1})$ be the output. For our example, for random evaluation points $\alpha_1 = 45, \alpha_2 = 6$ and $\alpha_3 = 69$ we obtain $V = (87, 26, 15, 94, 63, 15, 49, 74, 43, 71)$.

Now we use the Berlekamp/Massey algorithm [55] (See [41] for a more accessible reference). The input to this algorithm is a sequence of elements $s_0, s_1, \ldots, s_{2t-1}, \ldots$ from any field $\mathbb{F}$. If this sequence has a *linear generator* $\Lambda(z) = z^t - \lambda_{t-1}z^{t-1} - \cdots - \lambda_0$ of degree t, the algorithm computes it after processing $2t$ elements of the sequence. We have

$$s_{t+i} = \lambda_{t-1}s_{t+i-1} + \lambda_{t-2}s_{t+i-2} + \cdots + \lambda_0 s_i \text{ for all } i \geq 0.$$

In our example where $F = \mathbb{Z}_p$, the input is $V = (v_0, \ldots, v_{2T-1})$ and the output is

$$\Lambda_1(z) = z^5 + 80\,z^4 + 84\,z^3 + 16\,z^2 + 74\,z + 48.$$

In the next step, we choose n non-zero $(b_1, \ldots, b_n) \in \mathbb{Z}_p^n$ at random such that $b_k \neq \alpha_k$ for all $1 \leq k \leq n$. In this example we choose $b_1 = 44, b_2 = 9, b_3 = 18$. Now we choose the evaluation points $(b_1^i, \alpha_2^i, \ldots, \alpha_n^i)$ for $0 \leq i < 2T - 1$. Note that this time we are evaluating the first variable at powers of b_1 instead of α_1. We evaluate the black box at these points and apply the Berlekamp/Massey algorithm on the sequence of the outputs to compute the second linear generator

$$\Lambda_2 = z^5 + 48\,z^4 + 92\,z^3 + 9\,z^2 + 91\,z + 62.$$

We repeat the above process for a new set of evaluation points $(\alpha_1^i, b_2^i, \alpha_3^i, \ldots, \alpha_n^i) \in \mathbb{Z}_p^n$, i.e., we replace α_2 by b_2 obtaining

$$\Lambda_3 = z^5 + 42\,z^4 + 73\,z^3 + 73\,z^2 + 73\,z + 41$$

the third linear generator. Similarly, for evaluation points $(\alpha_1^i, \alpha_2^i, b_3^i)$ we compute

$$\Lambda_4 = z^5 + 73\,z^4 + 8\,z^3 + 94\,z^2 + 68\,z + 59.$$

Note that we can compute $\Lambda_1, \ldots, \Lambda_{n+1}$ in parallel. We know (see [3]) that if the monomial evaluations are distinct over $\mathbb{Z}_p$ for each set of evaluation points, then $\deg_z(\Lambda_i) = t$ for all $1 \le i \le n$ and each Λ_i has t non-zero roots in $\mathbb{Z}_p$. Ben-Or and Tiwari prove that for each $1 \le i \le t$, there exists $1 \le j \le t$ such that

$$m_i = M_i(\alpha_1, \ldots, \alpha_n) \equiv r_{0j} \bmod p.$$

where $r_{01}, \ldots, r_{0t}$ are the roots of Λ_1. In the next step we compute $r_{(i-1)1}, \ldots, r_{(i-1)t}$ the roots of the Λ_i. We have

$$\{r_{01} = 1, r_{02} = 50, r_{03} = 84, r_{04} = 91, r_{05} = 98\} \text{ (roots of } \Lambda_1)$$
$$\{r_{11} = 1, r_{12} = 10, r_{13} = 69, r_{14} = 84, r_{15} = 91\} \text{ (roots of } \Lambda_2)$$
$$\{r_{21} = 1, r_{22} = 25, r_{23} = 69, r_{24} = 75, r_{25} = 91\} \text{ (roots of } \Lambda_3)$$
$$\{r_{31} = 1, r_{32} = 8, r_{33} = 25, r_{34} = 35, r_{35} = 60\} \text{ (roots of } \Lambda_4)$$

The main step now is to determine the degrees of each monomial M_i of f in each variable. Consider the first variable x. We know that $m_i' = M_i(b_1, \alpha_2, \ldots, \alpha_n)$ is a root of Λ_2 for $1 \le i \le n$. On the other hand we have

$$\frac{m_i'}{m_i} = \frac{M_i(b_1, \alpha_2, \ldots, \alpha_n)}{M_i(\alpha_1, \alpha_2, \ldots, \alpha_n)} = (\frac{b_1}{\alpha_1})^{e_{i1}}. \tag{2.1}$$

Let $r_{0j} = M_i(\alpha_1, \alpha_2, \ldots, \alpha_n)$ and $r_{1k} = M_i(b_1, \alpha_2, \ldots, \alpha_n)$. From Equation 2.1 we have

$$r_{1k} = r_{0j} \times (\frac{b_1}{\alpha_1})^{e_{i1}},$$

i.e. for every root r_{0j} of Λ_1, $r_{0j} \times (\frac{b_1}{\alpha_1})^{e_{i1}}$ is a root of Λ_2 for some e_{i1} which is the degree of some monomial in f with respect to x. This gives us a way to compute the degree of each monomial M_i in the variable x.

In this example we have $\frac{b_1}{\alpha_1} = 93$. We start with the first root of Λ_1 and check if $r_{01} \times (\frac{b_1}{\alpha_1})^i$ is a root of Λ_2 for $0 \leq i \leq d$. To do this one could simply evaluate the polynomial $\Lambda_2(z)$ at $z = r_{01}(\frac{b_1}{\alpha_1})^i$ and see if we get zero. This costs $O(t)$ operations for each i. Instead we compute and sort the roots of Λ_2 so we can do this using binary search in $O(\log t)$. For $r_{01} = 1$ we have $r_{01} \times (\frac{b_1}{\alpha_1})^0 = 1$ is a root of Λ_2, and, for $0 < i \leq d$, $r_{01} \times (\frac{b_2}{\alpha_1})^i$ is *not* a root of Λ_2, hence we conclude that the degree of the first monomial of f in x is 0. We continue this to find the degrees of all the monomials in f in the variable x. We obtain

$$e_{11} = 0, e_{21} = 2, e_{31} = 0, e_{41} = 0, e_{51} = 2.$$

We proceed to the next variable y. Again using the same approach as above, we find that the degrees of the monomials in the second variable y to be

$$e_{12} = 0, e_{22} = 1, e_{32} = 1, e_{42} = 0, e_{52} = 2.$$

Similarly we compute the degrees of other monomials in z:

$$e_{13} = 0, e_{23} = 1, e_{33} = 2, e_{43} = 5, e_{53} = 1.$$

At this point we have computed all the monomials. Recall that $M_i = x_1^{e_{i1}} \times x_2^{e_{i2}} \times \cdots \times x_n^{e_{in}}$ hence we have

$$M_1 = 1, M_2 = x^2yz, M_3 = yz^2, M_4 = z^5, M_5 = x^2y^2z.$$

The reader may observe that once $\Lambda_1(z)$ is computed, determining the degrees of the monomials M_i in each variable represent n independent tasks which can be done in parallel. This is a key advantage of our algorithm.

Now we need to compute the coefficients. We do this by solving one linear system. We computed the roots of Λ_1 and we have computed the monomials such that $M_i(\alpha_1, \ldots, \alpha_n) = r_{0i}$. Recall that v_i is the output of the black box on input $(\alpha_1^i, \ldots, \alpha_n^i)$ hence we have

$$v_i = C_1 r_{01}^i + C_2 r_{02}^i + \cdots + C_t r_{0t}^i$$

for $0 \leq i \leq 2t - 1$. This linear system is a Vandermonde system which can be solved in $O(t^2)$ time and $O(t)$ space (see [85]). After solving we obtain

$$C_1 = 1, C_2 = 94, C_3 = 91, C_4 = 42 \text{ and } C_5 = 61$$

and hence $g = 1 + 94x^2yz + 91yz^2 + 42z^5 + 61x^2y^2z$ is our interpolated polynomial. We will show later that for p sufficiently large, $g = f$ with high probability. However, we can also check whether $g = f$ with high probability as follows; we choose evaluation points $(\alpha_1, \ldots, \alpha_n)$ at random and test if $\mathbf{B}(\alpha_1, \ldots, \alpha_n) = g(\alpha_1, \ldots, \alpha_n)$. If the results match, the algorithm returns g as the interpolated polynomial, otherwise it fails.

2.2 Problems

The evaluation points $\alpha_1, \ldots, \alpha_n$ must satisfy certain conditions for our new algorithm to output f. Here we identify all problems.

2.2.1 Distinct Monomials

The first condition is that for $1 \leq i \neq j \leq t$

$$M_i(\alpha_1, \ldots, \alpha_n) \neq M_j(\alpha_1, \ldots, \alpha_n) \text{ in } \mathbb{Z}_p$$

so that $\deg(\Lambda_1(z)) = t$. Also, at the k'th step of the algorithm, when computing the degrees of the monomials in x_k, we must have for all $1 \leq i \neq j \leq t$

$$m_{i,k} \neq m_{j,k} \text{ in } \mathbb{Z}_p \text{ where } m_{i,k} = M_i(\alpha_1, \ldots, \alpha_{k-1}, b_k, \alpha_{k+1}, \ldots, \alpha_n)$$

so that $\deg(\Lambda_{k+1}(z)) = t$. We now give an upper bound on the probability that no monomial evaluations collide when we use random non-zero elements of $\mathbb{Z}_p$ for evaluations.

Lemma 2.2. Let $\alpha_1, \ldots, \alpha_n$ be random non-zero evaluation points in $\mathbb{Z}_p$ and let $m_i = M_i(\alpha_1, \ldots, \alpha_n)$. Then the probability that two different monomials evaluate to the same value (we get a collision) is

$$\text{Prob}\big\{(m_i = m_j : \text{for some } 1 \leq i \neq j \leq t)\big\} \leq \binom{t}{2}\frac{d}{(p-1)} < \frac{dt^2}{2(p-1)}.$$

Proof. Consider the polynomial

$$A = \prod_{1 \leq i < j \leq t} \left(M_i(x_1, \ldots, x_n) - M_j(x_1, \ldots, x_n)\right).$$

Observe that $A(\alpha_1, \ldots, \alpha_n) = 0$ iff two monomial evaluations collide. Recall that the Schwartz-Zippel lemma (Lemma 1.7) says that if $r_1, \ldots, r_n$ are chosen at random from any subset S of a field K and $F \in K[x_1, \ldots, x_n]$ is non-zero then

$$\mathrm{Prob}(F(r_1, \ldots, r_n) = 0) \leq \frac{\deg F}{|S|}.$$

Our result follows from noting that $d \geq \deg f$ and thus $\deg A \leq \binom{t}{2} d$ and $|S| = p - 1$ since we choose α_i to be non-zero from $\mathbb{Z}_p$. □

Remark 2.3. If any of the $\alpha_i = 1$ then the probability of monomial evaluations colliding is clearly high. To reduce the probability of monomial evaluations colliding, in an earlier version of our algorithm, we picked α_i to have order $> d$. We did this by picking random generators of $\mathbb{Z}_p^*$. There are $\phi(p-1)$ generators where ϕ is Euler's totient function. However, if one does this, the restriction on the choice of α leads to a weaker result, namely, $\binom{t}{2} \frac{d}{\phi(p-1)}$.

2.2.2 Root Clashing

Let $r_{01}, \ldots, r_{0t}$ be the roots of $\Lambda_1(z)$ which is the output of the Berlekamp/Massey algorithm using the first set of evaluation points $(\alpha_1^i, \ldots, \alpha_n^i)$ for $0 \leq i < 2T$. To compute the degrees of all the monomials in the variable x_k, as mentioned in the Example 2.1, the first step is to compute Λ_{k+1}. Then if $\deg_{x_k}(M_i) = e_{ik}$ we have $r_{ki} = r_{0i} \times (\frac{b_k}{\alpha_k})^{e_{ik}}$ is a root of Λ_{k+1}. If $r_{0i} \times (\frac{b_k}{\alpha_k})^{e'}$, $0 \leq e' \neq e_{ik} \leq d$ is also a root of Λ_{k+1} then we have a root clash and we cannot uniquely identify the degree of the monomial M_i in x_k.

Example 2.4. Consider the polynomial given in Example 2.1. Suppose instead of choosing $b_1 = 44$, we choose $b_1 = 72$. Since α_1, α_2 and α_3 are the same as before, Λ_1 does not change and hence the roots of Λ_1 are $r_{01} = 1, r_{02} = 7, r_{03} = 41, r_{04} = 61$ and $r_{05} = 64$. In the next step we substitute $b_1 = 72$ for α_1 and compute $\Lambda_2 = z^5 + 61z^4 + 39z^3 + 67z^2 + 37z + 98$. We proceed to compute the degrees of the monomials in x but we find that

$$r_4 \times (\frac{\alpha_4}{\alpha_1})^2 = 15 \;\text{ and }\; r_4 \times (\frac{\alpha_4}{\alpha_1})^4 = 7$$

are both roots of Λ_2, hence we can not determine the degree of the last monomial in x.

Lemma 2.5. If $\deg \Lambda_1(z) = \deg \Lambda_{k+1}(z) = t$ then the probability that there is a root clash, that is, we can not uniquely compute the degrees of all the monomials $M_i(x_1, \ldots, x_n)$ in x_k is at most $\frac{d(d+1)t^2}{4(p-2)}$.

Proof. Let $S_i = \{r_{0j} \times (\frac{b_k}{\alpha_k})^i \mid 1 \leq j \leq t\}$ for $0 \leq i \leq d$. We assume that $r_{0i} \neq r_{0j}$ for all $1 \leq i \neq j \leq t$. We will not be able to uniquely identify the degree of the j'th monomial in x_k if there exists $\bar{d}$ such that $r_{0j} \times (\frac{b_k}{\alpha_k})^{\bar{d}} = r_{ki}$ is a root of $\Lambda_{k+1}(z)$ and $0 \leq d \neq e_{jk} \leq d$ where e_{jk} is $\deg_{x_k}(M_j)$. But we have $r_{ki} = r_{0i} \times (\frac{b_k}{\alpha_k})^{e_{ik}}$ thus $r_{0j} \times (\frac{b_k}{\alpha_k})^{\bar{d}} = r_{0i} \times (\frac{b_k}{\alpha_k})^{e_{ik}}$. Without loss of generality, assume $\tilde{d} = \bar{d} - e_{ik} > 0$. We have $r_{0i} = r_{0j} \times (\frac{b_k}{\alpha_k})^{\tilde{d}}$ and hence $r_{0i} \in S_{\tilde{d}} \Rightarrow S_0 \cap S_{\tilde{d}} \neq \emptyset$. Hence we will not be able to compute the degrees in x_k if $S_0 \cap S_i \neq \emptyset$ for some $1 \leq i \leq d$. Let

$$g(x) = \prod_{1 \leq l \neq j \leq t} (r_{0j} x^i - r_{0l} \alpha_k^i).$$

We have $r_{0l} = r_{0j} \times (\frac{b_k}{\alpha_k})^i \in S_0 \cap S_i$ iff $g(b_k) = 0$. Applying the Schwartz-Zippel lemma (Lemma 1.7), the probability that $g(b_k) = 0$ is at most $\frac{\deg g}{|S|} = \frac{\binom{t}{2} i}{(p-2)} < \frac{it^2}{2(p-2)}$ since we chose $b_k \neq \alpha_k \neq 0$ at random from $\mathbb{Z}_p$. If we sum this quantity for all $1 \leq i \leq d$ we obtain that the overall probability is at most $\frac{d(d+1)t^2}{4(p-2)}$. □

2.3 The Algorithm

ALGORITHM 2.1: Algorithm: **Parallel Interpolation**

```
Require: A black box $\mathbf{B} : \mathbb{Z}_p^n \to \mathbb{Z}_p$ that on input $\alpha_1, \ldots, \alpha_n \in \mathbb{Z}_p^n$ outputs $f(\alpha_1, \ldots, \alpha_n)$
    where $f \in \mathbb{Z}_p[x_1, \ldots, x_n] \backslash \{0\}$.
Require: A degree bound $d \geq \deg(f)$.
Require: A bound $T \geq t$ on the number of terms in $f$. {For reasonable probability of
    success we require $p > dT^2$.}
Ensure: The polynomial $f$ or FAIL.
Choose $\alpha_1, \ldots, \alpha_n$ from $\mathbb{Z}_p \backslash \{0\}$ at random.
for $k$ from 0 to $n$ in parallel do
    Case $k = 0$ : Compute $\Lambda_1(z)$ using $(\alpha_1, \ldots, \alpha_n)$:
    Evaluate the black box $\mathbf{B}$ at $(\alpha_1^i, \ldots, \alpha_n^i) \in \mathbb{Z}_p^n$ for $0 \leq i \leq 2T - 1$ and apply the
    Berlekamp Massey algorithm to the sequence of $2T$ outputs.
    Case $k > 0$: Choose non-zero $b_k \in \mathbb{Z}_p \backslash \{0\}$ at random until $b_k \neq \alpha_k$ and
    compute $\Lambda_{k+1}(z)$ using $\alpha_1, \ldots, a_{k-1}, b_k, \alpha_{k+1}, \ldots, \alpha_n$.
end for
```

```
6: Set $t = \deg \Lambda_1(z)$. If the degree of the $\Lambda$'s are not all equal to $t$ then **return** FAIL.
7: **for** $k$ from 0 to $n$ in **parallel do**
8:     Compute $\{r_{k1}, \ldots, r_{kt}\}$ the set of distinct roots of $\Lambda_{k+1}(z)$.
9: **end for**
10: **for** $k$ from 1 to $n$ in **parallel do**
11:     Determine $\deg_{x_k}(M_i)$ for $1 \le i \le t$ as described in Section 2.1. If we failed to to
        compute the degrees uniquely (see Section 2.2.2) then **return** FAIL.
12: **end for**
13: Let $S = \{C_1 r_{01}^i + C_2 r_{02}^i + \cdots + C_t r_{0t}^i = v_i \mid 0 \le i \le 2t-1\}$. Solve the linear system $S$
    for $(C_1, \ldots, C_t) \in \mathbb{Z}_p^t$ and set $g = \sum_{i=1}^t C_i M_i$ where $M_i = \prod_{j=1}^n x_j^{e_{ij}}$.
14: Choose $\alpha_1, \ldots, \alpha_n$ from $\mathbb{Z}_p \backslash \{0\}$ at random.
    If $\mathbf{B}(\alpha_1, \ldots, \alpha_n) \ne g(\alpha_1, \ldots, \alpha_n)$ then **return** FAIL.
15: **return** $g$.
```

Remark 2.6. The algorithm presented corresponds to our parallel implementation in Cilk. Further parallelism is available. In particular, one may compute the $2T$ probes to the black box $\mathbf{B}$ in step 3 in parallel. We remark that Kaltofen in [37] pointed out to us that assuming $T \ge t$, then $T+t$ probes are sufficient to determine any $\Lambda(z)$ and that the Berlekamp-Massey algorithm can be modified to stop after processing $T+t$ inputs.

Remark 2.7. The algorithm is probabilistic. If the degrees of the Λ's are all equal but less than t then monomial evaluations have collided and the algorithm cannot compute f. The check in step 14 detects incorrect g with probability at least $1 - d/(p-1)$ (Lemma 1.7). Thus by doing one additional probe to the black box, we verify the output g with high probability. Kaltofen and Lee in [40] also use additional probes to verify the output this way.

Theorem 2.8. If $p > 2^{k-2}3(n+1)d(d+3)t^2 + 2$ then Algorithm Parallel Interpolation outputs $f(x_1, \ldots, x_n)$ with probability at least $1 - 1/2^k$. Moreover, the probability that the algorithm outputs an incorrect result is less than $\frac{d}{p-1} \times \left(\frac{dt^2}{2(p-1)}\right)^n$.

Proof. Algorithm Parallel Interpolate will need $(n+1)$ Λ's all of degree t. The choice of α_k, and b_k must be non-zero and distinct. Thus applying Lemmas 2.2 and 2.5, the probability that all $n+1$ Λ's have degree t and we can compute all the monomial degrees with no collisions is at least $1 - \frac{(n+1)dt^2}{2(p-2)} - \frac{n\,d(d+1)t^2}{4(p-2)} > 1 - \frac{3(n+1)d(d+3)t^2}{4(p-2)}$. Solving $1 - \frac{3(n+1)d(d+3)t^2}{4(p-2)} >$

$1 - 2^{-k}$ for $p - 2$ gives the first result. For the second result, the algorithm outputs an incorrect result only if all $\Lambda_1, \ldots, \Lambda_{n+1}$ have degrees less than t and the check in step 14 fails. This happens with probability less than $(\frac{dt^2}{2(p-1)})^n$ (see Lemma 2.2) and less than $\frac{d}{p-1}$ (see Remark 2.7), respectively. □

2.3.1 Complexity Analysis

We now give the sequential complexity of the algorithm in terms of the number of arithmetic operations in $\mathbb{Z}_p$. We need to consider the cost of probing the black box. Let $E(n, t, d)$ be the cost of one probe to the black box. We make $2(n+1)T$ probes in the first loop and one in step 14. Hence the cost of probes to the black box is $O(nTE(n, t, d))$.

The $n + 1$ calls to the Berlekamp/Massey algorithm in the first loop (as presented in [41]) cost $O(T^2)$ each. The Vandermonde system of equations at Step 13 can be solved in $O(t^2)$ using the method given in [85]. Note that as mentioned in [85], when inverting a $t \times t$ Vandemonde matrix defined by $k_1, \ldots, k_t$, one of the most expensive parts is to compute the master polynomial $M(z) = \prod_{i=1}^{t}(z - k_i)$. However, in our algorithm we can use the fact that $M(z) = \prod_{i=1}^{t}(z - r_{0i}) = \Lambda_1(z)$.

To compute the roots of $\Lambda_{k+1}(z)$ at Step 8, we use Rabin's Las Vegas algorithm from [63]. The idea of Rabin's algorithm is to split $\Lambda(z)$ using the following gcd in $\mathbb{Z}_p[z]$

$$g(x) = \gcd((z + \beta)^{(p-1)/2} - 1, \Lambda(z))$$

for β chosen at random from $\mathbb{Z}_p$. For $\Lambda(z)$ with degree t, if classical algorithms for polynomial multiplication, division and gcd are used for $\mathbb{Z}_p[z]$, the cost is dominated by the first split which has expected cost $O(t^2 \log p)$ (see Ch. 8 of Geddes et. al. [18]) arithmetic operations in $\mathbb{Z}_p$.

To compute the degree of the monomials in the variable x_k in Step 11 of the algorithm, we sort the roots of $\Lambda_1(z)$ and $\Lambda_{k+1}(z)$. Then checking if $r_{0i} \times (\frac{b_k}{\alpha_k})^{\bar{d}}$ is a root of $\Lambda_{k+1}(z)$ can be done in $O(\log t)$ using binary search. Hence the the degrees can be computed in $O(t \log t + dt \log t)$.

Theorem 2.9. The expected number of arithmetic operations in $\mathbb{Z}_p$ for a sequential run of our new algorithm is

$$O(n(t^2 \log p + dt \log t + T^2 + TE(n, t, d))$$

using classical (quadratic) algorithms for polynomial arithmetic. For $T \in O(t)$ this simplifies to $O(n(t^2 \log p + dt \log t + tE(n,t,d))$.

Apart from the cost of the probes, the most expensive component of the algorithm is the computation of the roots of the $\Lambda(z)$'s, each of which costs $O(t^2 \log p)$ using classical arithmetic. It is well known that this can be improved to $O(\log t(M(t)\log p + M(t)\log t))$ using fast multiplication (see Algorithm 14.15 of von zur Gathen and Gerhard [73]) where $M(t)$ is cost of multiplication of polynomials of degree t in $\mathbb{Z}_p[z]$. In our implementation Michael Monagan implemented this asymptotically fast root finding algorithm because we found that the root finding was indeed an expensive component of the cost.

Similarly, the generator polynomial $\Lambda_k(z)$ can also be computed using the Fast Euclidean Algorithm in $O(M(t)\log t)$. See [73] Ch. 11 for a description of the Fast Euclidean Algorithm and [73] Ch. 7 for a description of how to compute $\Lambda_k(z)$. Furthermore, once the support (the monomials) for a polynomial are known, the coefficients can be determined in $O(M(t)\log t)$ time using fast multiplication (see van der Hoven and Lecerf [26]). This leads to a complexity, softly linear in T, of $O(n(M(t)\log p \log t + dt \log t + M(T)\log T + TE(n,t,d)))$.

If we choose p to be a Fourier prime then $M(t) \in O(t \log t)$ using the FFT. Hence the expected sequential complexity of our algorithm is $O(n[t \log^2 t \log p + dt \log t + T \log^2 T + TE(n,t,d)])$ arithmetic operations in $\mathbb{Z}_p$.

Zippel's Algorithm

Recall from Chapter 1 that the number of probes to the black box for Zippel's algorithm is $O(ndt)$. However we expect Zippel's algorithm to perform better than our algorithm for *dense* target polynomials.

Lemma 2.10. Let f be a dense polynomial of degree d in each variable so that the number of terms in f is $t = (d+1)^n$. Then the number of probes to the black box in Zippel's algorithm is exactly t.

Proof. Here we have $t_i = (d+1)^i$ thus the number of probes is $1 + d \times \prod_{i=0}^{n-1}(d+1)^i = 1 + d \times \frac{(d+1)^n - 1}{d+1-1} = (d+1)^n = t$. □

In comparison, our algorithm does $2(n+1)t + 1$ probes.

2.3.2 Optimizations

Computing the degrees of the monomials in the last variable.

The first optimization is to compute the degree of each monomial $M_i = x_1^{e_{i1}} x_2^{e_{i2}} \dots x_n^{e_{in}}$ in the last variable x_n without doing any more probes to the black box. Suppose we have computed the degree of M_i in x_k for $1 \leq k < n$. We know that $M_i(\alpha_1, \dots, \alpha_n)$ is equal to r_{0i}, a root of Λ_1. Hence $r_{0i} = \alpha_1^{e_{i1}} \cdot \alpha_2^{e_{i2}} \cdot \dots \cdot \alpha_n^{e_{in}}$. Since we know the degrees e_{ij} for $1 \leq j < n$ we can determine e_{in} by division of $r_{0i} \cdot (\alpha_1^{e_{i1}} \dots \alpha_{n-1}^{e_{in-1}})^{-1}$ by α_n. This reduces the total number of probes from $2(2n+1)t$ to $2(2n-1)t$ and increases the probability of success from $> 1 - \frac{3(n+1)d(d+3)t^2}{4(p-2)}$ to $> 1 - \frac{3nd(d+3)t^2}{4(p-2)}$.

Bipartite perfect matching.

We now present an improvement that will allow our algorithm to determine the degree of the monomial M_i in x_k even when $r_{0i} \times \frac{b_k}{\alpha_k}^{e'}$ is also a root of $\Lambda_{k+1}(z)$ in most cases. Note that we assume the monomial evaluations are distinct, i.e. $\forall\ 1 \leq i \neq j \leq t,\ m_{i,k} \neq m_{j,k}$.

Suppose we have computed Λ_{k+1} and we want to compute the degrees of the monomials in x_k and let $R_1 = \{r_{01}, \dots, r_{0t}\}$ be the set of roots of Λ_1 and $R_k = \{r_{k1}, \dots, r_{kt}\}$ be the set of roots of Λ_{k+1}. Let

$$D_j = \{(i, r) \mid 0 \leq i \leq d,\ r = r_{0j} \times (\frac{b_k}{\alpha_1})^i \in R_k\}.$$

D_j contains the set of all possible degrees of the j'th monomial M_j in the k'th variable x_k. We know that $(e_{jk}, r_{kj}) \in D_j$ and hence $|D_j| \geq 1$. If $|D_j| = 1$ for all $1 \leq j \leq t$, then the degrees are unique and this step of the algorithm is complete. Let G_k be a balanced bipartite graph defined as follows. G_k has two independent sets of nodes U and V each of size t. Nodes in U and V represent elements in R_1 and R_k respectively, i.e. $u_i \in U$ and $v_j \in V$ are labeled with r_{0i} and r_{kj}. We connect $u_i \in U$ to $v_j \in V$ with an edge of weight (degree) d_{ij} if and only if $(d_{ij}, r_{kj}) \in D_i$. We illustrate with an example.

Example 2.11. Let f be the polynomial given in Example 2.1 and suppose for some evaluation points $\alpha_1, \dots, \alpha_3$ and b_1 we obtain the graph G_1 as shown in Figure 2.1. Notice that this graph has a unique perfect matching, i.e., the set of edges $\{(r_{0i}, r_{1i}) \mid 1 \leq i \leq 5\}$. Thus the degrees of the 5 monomials in x must be are 0, 0, 0, 2, and 2.

Lemma 2.12. We can uniquely identify the degrees of all the monomials in x_k if the bipartite graph G_k has a unique *perfect matching*.

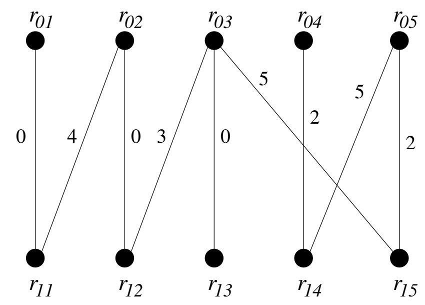


Figure 2.1: The bipartite graph G_1

Proof. Let $m_i = M_i(\alpha_1, \ldots, \alpha_n)$ and without loss of generality, assume $r_{0i} = m_i$ and $r_{ki} = m_{i,k}$ for all $1 \leq i \leq t$. We have $(e_{ik}, r_{ki}) \in D_i$ and hence $u_i \in U$ is connected to $v_j \in V$ in G_k with an edge of weight e_{ik}. This means that the set $S = \{(u_i, v_i, e_{ik}) \mid u_i \in U, v_i \in V\}$ is a perfect matching in G_k. If this perfect matching is unique then by finding it, we have computed e_{ik}'s, the degrees of the monomials in x_k. □

To find a prefect matching in the graph G_k one can use the Hopcroft–Karp algorithm [28]. This algorithm finds a matching in time $O(e\sqrt{v})$ where e and v are the number of edges and vertices respectively. However, for random sparse bipartite graphs, Bast *et al.* [2] (See also [60]) prove that the Hopcroft-Karp algorithm runs in time $O(e \log v)$ with high probability.

Lemma 2.13. If $p > dT^2 + 1$ then the expected number of edges in the graph G_k is at most

$$\frac{d+1}{4} + t.$$

Proof. In lemma 2.5 we showed that the probability that there are no root clashes is greater than $1 - \frac{d(d+1)t^2}{4(p-2)}$. Therefore, the expected number of root clashes is less than $\frac{d(d+1)t^2}{4(p-2)}$ hence the expected number of edges of G_k is less than $t + \frac{d(d+1)t^2}{4(p-2)}$. If we choose p such that $p - 1 > dT^2$ then $p - 1 > dt^2$ and the expected number of edges in G_k is at less than $(d+1)/4 + t$. □

Thus if $4(d+1) \leq t$ then the expected number of edges is at most $2t$, hence G_k is sparse and the expected cost of finding a perfect matching would be $O(t \log t)$, which is softly linear in t.

Doing additional probes to get a perfect matching.

We mentioned that if the graph G_k does not have a unique perfect matching, the algorithm fails. We now give a solution for this. The solution involves $2t$ more probes to the black box. Suppose we choose a random element $c_k \in \mathbb{Z}_p$ such that $\gamma = \frac{c_k}{b_k}$ is of order greater than d. Let $\beta_i = (\alpha_1^i, \ldots, \alpha_{k-1}^i, c_k^i, \alpha_{k+1}^i, \ldots, \alpha_n^i)$ and let v_i be the output of the black box on input β_i $(0 \le i \le 2t-1)$. On input $V = (\beta_0, \ldots, \beta_{2t-1})$, the Berlekamp/Massey algorithm computes a linear generator $\Lambda'_{k+1}(z)$ for V. Let $\{\tilde{r}_{k1}, \ldots, \tilde{r}_{kt}\}$ be the set of distinct roots of Λ'_{k+1}. Let G'_k be the balanced bipartite graph, obtained from Λ_1 and Λ'_{k+1}.

Definition 2.14. We define $\bar{G}_k$, the intersection of G'_k and G_k, as follows. $\bar{G}_k$ has the same nodes as G'_k and there is an edge between r_{0i} and $\tilde{r}_{kj}$ with weight (degree) d_{ij} if and only if r_{0i} is connected to r_{kj} in G_k and to $\tilde{r}_{kj}$ in G'_k, both with the same degree d_{ij}.

Lemma 2.15. Let $e_{ij} = \deg_{x_j}(M_i)$. The two nodes r_{0i} and $\tilde{r}_{ki}$ are connected in $\bar{G}_k$ with degree e_{ij}.

We take advantage of the following theorem which implies we need at most one extra set of probes.

Theorem 2.16. Let $\bar{G}_k = G_k \cap G'_k$. $\bar{G}_k$ has a *unique* perfect matching.

Proof. Let U and V be the set of independent nodes in $\bar{G}_k$ such that $u_i \in U$ and $v_j \in V$ are labeled with r_{0i} and $\tilde{r}_{kj}$ respectively where $\tilde{r}_{kj}$ is a root of Λ'_{k+1}. We will prove that each node in V has degree exactly 1 and hence there is a unique perfect matching. The proof is by contradiction. Suppose the degree of $v_j \in V$ is at least 2. With out loss of generality assume that r_{01} and r_{02} are both connected to $\tilde{r}_{kj}$ with degrees d_{1j} and d_{2j} respectively (See Figure 2.2).

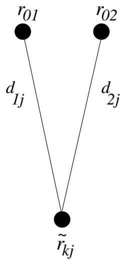


Figure 2.2: Node $\tilde{r}_{kj}$ of graph $\bar{G}_k$

Using Definition 2.14 we have

$$r_{kj} = r_{01} \times (\frac{b_k}{\alpha_k})^{d_{1j}} = r_{02} \times (\frac{b_k}{\alpha_k})^{d_{2j}} \text{ and}$$

$$\tilde{r}_{kj} = r_{01} \times (\frac{c_k}{\alpha_k})^{d_{1j}} = r_{02} \times (\frac{c_k}{\alpha_k})^{d_{2j}}.$$

Dividing the two sides of these equations results in

$$(\frac{c_k}{b_k})^{d_{1j}} = (\frac{c_k}{b_k})^{d_{2j}}.$$

Since we chose c_k such that $\frac{c_k}{b_k}$ has a sufficiently large order (greater than the degree bound d) we have $d_{1j} = d_{2j} \Rightarrow r_{01} = r_{02}$. But this is a contradiction because both r_{01} and r_{02} are roots of Λ_1 which we assumed are distinct. □

Lemma 2.15 and Theorem 2.16 prove that the intersection of G_k and G'_k will give us the correct degrees of all the monomials in the k'th variable x_k. We will illustrate with an example.

Example 2.17. Let $f = -10\, y^3 - 7\, x^2yz - 40\, yz^5 + 42\, y^3z^5 - 50\, x^7z^2 + 23\, x^5z^4 + 75\, x^7yz^2 - 92\, x^6y^3z + 6\, x^3y^5z^2 + 74\, xyz^8 + 4$ and $p = 101$. We choose the first set of evaluation points to be $\alpha_1 = 66, \alpha_2 = 11, \alpha_3 = 48$ and $b_1 = 50$. For the first variable x we will obtain the bipartite graph G_1 shown in Figure 2.3.

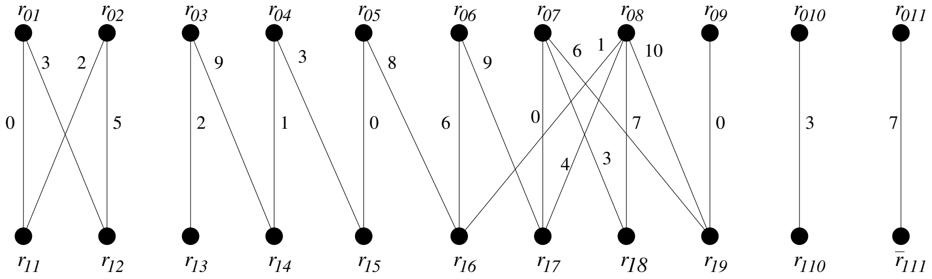


Figure 2.3: The bipartite graph G_1

This graph does not have a unique perfect matching, so we proceed to choose a new evaluation point $c_1 = 89$. This time we will get the bipartite graph G'_1 shown in Figure 2.4.

Again G'_1 does not have a unique perfect matching. We compute the intersection of G_1 and G'_1: $\bar{G}_1 = G_1 \cap G'_1$. $\bar{G}_1$ is shown in Figure 2.5.

As stated by Theorem 2.16, $\bar{G}_1$ has a unique perfect matching and the degree of every monomial in x is correctly computed.

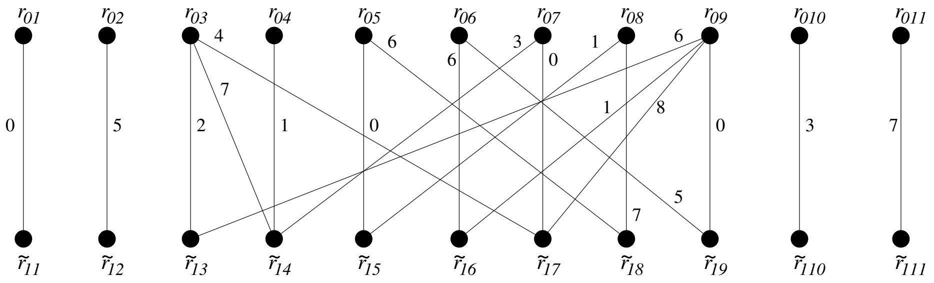


Figure 2.4: The bipartite graph G'_1

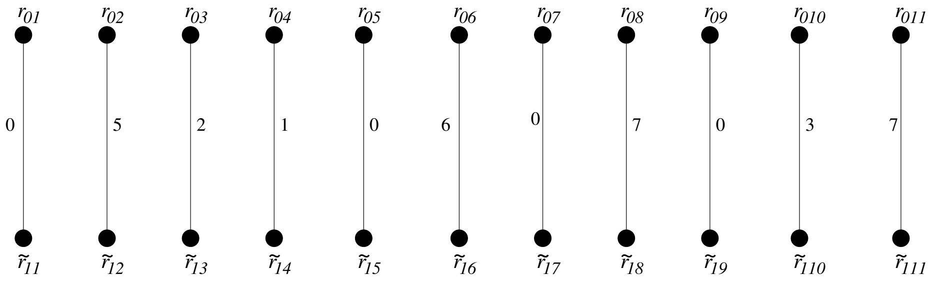


Figure 2.5: The bipartite graph $\bar{G}_1$

In this section we proved that if the prime p is sufficiently large (p must be approximately ndt^2 for us to be able to get distinct images of monomials with reasonable probability), we will be able to compute the degrees of all the t monomials in each variable x_k using up to $4t$ evaluation points. If the graph G_k has a unique perfect matching, we will be able to compute the degrees in x_k with only $2T$ probes to the black box.

Computing the degrees of the monomials in x_k.

Let $D = \deg(f)$. If the prime p is large enough, i.e. $p > \frac{nD(D+1)t^2}{4\epsilon}$ then with probability $1-\epsilon$ the degree of every monomial in x_k can correctly be computed using only G_k and without needing any extra probes to the black box. In fact in this case, with high probability, every r_{0i} will be matched with exactly only one r_{kj} and hence every node in G_k would have degree one. But if $d \gg D$, i.e. the degree bound d is not tight, the probability that we could identify the degrees uniquely drops significantly even though p is large enough. This is because the probability that *root clashing* (see Section 2.2) happens, linearly depends on d. In this case, with probability $1-\epsilon$, the degree of M_i in x_k would be $\min\{d_{ij} \mid (d_{ij}, r_i) \in G_k\}$, i.e. the edge connected to r_{0i} in G_k with minimum weight (degree) is our desired edge in the graph which will show up in the perfect matching. We apply Theorem 2.20 below.

Lemma 2.18. Let G_k be the bipartite graph for the k'th variable. Let $u_{i_1} \to v_{j_1} \to u_{i_2} \to v_{j_2} \to \cdots \to v_{j_s} \to u_{i_1}$ be a cycle in G_k where $u_l \in U$ is labeled with r_{0l} (a root of Λ_1) and $v_m \in V$ is labeled with r_{km} (a root of Λ_{k+1}). Let d_{lm} be the weight (degree) of the edge between u_l and v_m. We have $\sum_{m=1}^{s} d_{i_mj_m} - \sum_{m=1}^{s} d_{i_{m+1}j_m} = 0$.

Proof. It is easy to show that $r_{0i_1} = (\frac{b_k}{\alpha_k})^{\bar{d}} r_{0i_s}$ where $\bar{d} = d_{i_1j_1} - d_{i_2j_1} + d_{i_2j_2} - d_{i_3j_2} + \cdots + d_{i_{s-1}j_{s-1}} - d_{i_sj_{s-1}}$. Also both u_{i_1} and u_{i_s} are connected to v_{j_s} in G_k hence we have $r_{0i_1} = (\frac{b_k}{\alpha_k})^{d_{i_1j_s}} r_{ki_s}$ and $r_{0i_s} = (\frac{b_k}{\alpha_k})^{d_{i_sj_s}} r_{ki_s}$. These three equations yield to $r_{0i_1} = (\frac{b_k}{\alpha_k})^{\tilde{d}} r_{0i_1}$ where $\tilde{d} = d_{i_1j_1} - d_{i_2j_1} + d_{i_2j_2} - d_{i_3j_2} + \cdots + d_{i_{s-1}j_{s-1}} - d_{i_sj_{s-1}} + d_{i_sj_s} - d_{i_1j_s}$. But if $\frac{b_k}{\alpha_k}$ is of sufficiently high order, $\tilde{d}$ must be zero thus $\sum_{m=1}^{s} d_{i_mj_m} - \sum_{m=1}^{s} d_{i_{m+1}j_m} = 0$. □

Example 2.19. In G_1' shown in Figure 2.6, there is a cycle $r_3 \to \tilde{r}_4 \to r_7 \to \tilde{r}_7 \to r_3$. The weights (degrees) of the edges in this cycle are as $7, 3, 0$ and 4. We have $7 - 3 + 0 - 4 = 0$.

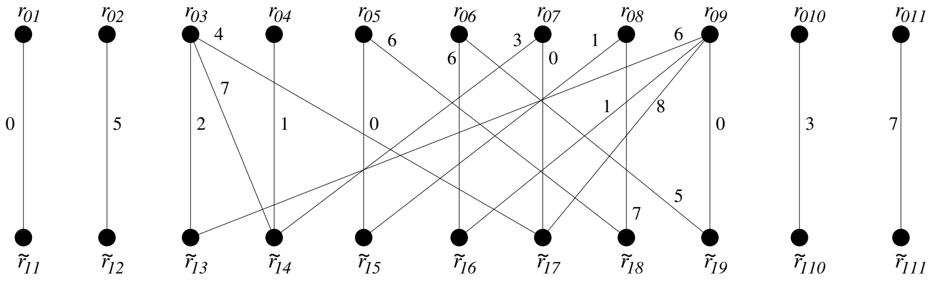
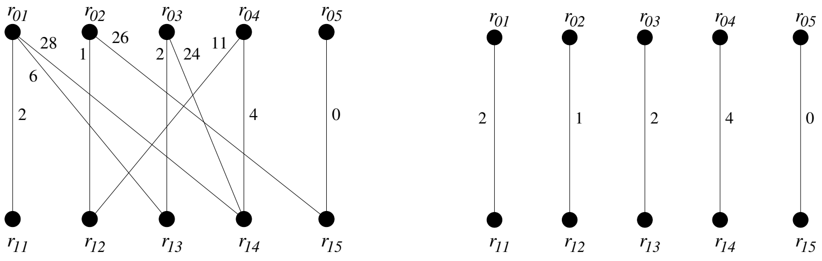


Figure 2.6: The bipartite graph G_1'

Theorem 2.20. Let H_k be a graph obtained by removing all edges connected to r_{0i} in G_k except the one with minimum weight (degree) for all $1 \leq i \leq t$. If the degree of every node in H_k is one, then e_{ik} is equal to the weight of the edge connected to r_{0i} in H_k.

This theorem can be proved using Lemma 2.18 and the fact that there can not be any cycle in the graph H_k. We give an example as follows.

Example 2.21. Let $f = 25y^2z + 90yz^2 + 93x^2y^2z + 60y^4z + 42z^5$. Here $t = 5, n = 3, \deg(f) = 5$ and $p = 101$. We choose the following evaluation points $\alpha_1 = 85, \alpha_2 = 96, \alpha_3 = 58$ and $b_1 = 99$. Suppose we want to construct G_2 in order to compute the degrees of the monomials in y. Suppose our degree bound is $d = 40$ which is not tight. The graph G_2 and H_2 are shown in Figures 2.7(a) and 2.7(b) respectively. The graph H_2 has the correct degrees of the monomials in y.



(a) G_2 (b) H_2

Figure 2.7: The bipartite graphs G_2 and H_2

Theorem 2.20 suggests the following optimization. In the construction of the bipartite graph G_k, connect r_{0i} to r_{kj} with degree d_{ij} *only if* there is no $\bar{d} < d_{ij}$ such that $r_{0i} \times (\frac{b_k}{\alpha_k})^{\bar{d}}$ is a root of Λ_{k+1}, i.e. the degree of the node r_{0i} in U is always one for all $1 \leq i \leq n$. If there is a perfect matching in this graph, this perfect matching is unique because this implies that the degree of each node r_{kj} in V is also one (e.g. see Figure 2.7(b)). If not, go back to and complete the graph G_k. This optimization makes our algorithm sensitive to the actual degree of $f(x_1, \ldots, x_n)$ in each variable.

2.4 Benchmarks

Here, we compare the performance of our new algorithm, Zippel's algorithm and the racing algorithm of Kaltofen and Lee from [40]. We have implemented Zippel's algorithm and our new algorithm in C. We have also implemented an interface to call the interpolation routines from Maple. The racing algorithm is implemented in Maple in the ProtoBox package by Lee [40]. Since this algorithm is not coded in C, we only report (see columns labeled ProtoBox) the number of probes it makes to the black box.

We give benchmarks comparing their performance on five problem sets. The polynomials in the first four benchmarks were generated at random. The fifth set of polynomials is taken from [40]. We count the number of probes to the black box that each algorithm takes and we measure the total CPU time for our new algorithm and Zippel's algorithm only. All timings reported are in CPU seconds and were obtained using Maple 13 on a 64 bit Intel Core i7 920 @ 2.66GHz running Linux. This is a 4 core machine. For our algorithm, we report the real time for 1 core and (in parentheses) 4 cores.

The black box in our benchmarks computes a multivariate polynomial with coefficients in $\mathbb{Z}_p$ where $p = 2114977793$ is a 31 bit prime. In all benchmarks, the black box simply evaluates the polynomial at the given evaluation point. To evaluate efficiently we compute and cache the values of $x_i^j \bmod p$ in a loop in $O(nd)$. Then we evaluate the t terms in $O(nt)$. Hence the cost of one black box probe is $O(nd + nt)$ arithmetic operations in $\mathbb{Z}_p$.

Remark 2.22. The second optimization described in Section 2.3.2 was not needed in any of these benchmarks. The algorithm can uniquely compute the degrees without requiring to do any bipartite matching.

Benchmark #1

This set of problems consists of 13 multivariate polynomials in $n = 3$ variables. The i'th polynomial ($1 \leq i \leq 13$) is generated at random using the following Maple command:

```
> randpoly([x1,x2,x3], terms = 2^i, degree = 30) mod p;
```

The i'th polynomial will have about 2^i non-zero terms. Here $D = 30$ is the total degree hence the maximum number of terms in each polynomial is $t_{max} = \binom{n+D}{D} = 5456$. We run both Zippel's algorithm and our new algorithm with degree bound $d = 30$. The timings and the number of probes are given in Table 2.1. In this table "DNF" means that the algorithm did not finish after 12 hours.

As i increases, the polynomial f becomes denser. For $i > 6$, f has more than $\sqrt{t_{max}}$ non-zero terms. This is indicated by a horizontal line in Table 2.1 and also in subsequent benchmarks. The line approximately separates sparse inputs from dense inputs. The last polynomial ($i = 13$) is 99.5% dense.

The data in Table 2.1 shows that for sparse polynomials $1 \leq i \leq 6$, our new algorithm does a lot fewer probes to the black box compared to Zippel's algorithm. It also does fewer probes than the racing algorithm (ProtoBox). However, as the polynomials get denser, Zippel's algorithm has a better performance. For a completely dense polynomial with t non-zero terms, Zippel's algorithm only does $O(t)$ probes to the black box while the new algorithm does $O(nt)$ probes.

To show how effective the first optimization described in Section 2.3.2 is, we run both our algorithm and Zippel's algorithm on the same set of polynomials but with a bad degree bound $d = 100$. The timings and the number of probes are given in Table 2.2. One can see

Table 2.1: benchmark #1: $n = 3$ and $D = 30$

i	t	New Algorithm		Zippel		ProtoBox
		Time	Probes	Time	Probes	Probes
1	2	0.00 (0.00)	13	0.00	217	20
2	4	0.00 (0.00)	25	0.00	341	39
3	8	0.00 (0.00)	49	0.00	558	79
4	16	0.00 (0.00)	97	0.01	868	156
5	32	0.00 (0.00)	193	0.01	1519	282
6	64	0.01 (0.00)	385	0.03	2573	517
7	128	0.02 (0.01)	769	0.08	4402	962
8	253	0.08 (0.03)	1519	0.21	6417	1737
9	512	0.17 (0.09)	3073	0.55	9734	3119
10	1015	0.87 (0.29)	6091	1.16	12400	5627
11	2041	3.06 (1.01)	12247	2.43	15128	DNF
12	4081	10.99 (3.71)	24487	4.56	16182	DNF
13	5430	19.02 (6.23)	32581	5.93	16430	DNF

that our algorithm is unaffected by the bad degree bound; the number of probes and CPU timings are the same.

Benchmark #2

In this set of benchmarks the i'th polynomial is in $n = 3$ variables and is generated at random in Maple using

```
> randpoly([x1,x2,x3], terms = 2^i, degree = 100) mod p;
```

This set of polynomials differs from the first benchmark in that the total degree of each polynomial is set to be 100 in the second set. We run both the Zippel's algorithm and our new algorithm with degree bound $d = 100$. The timings and the number of probes are given in Table 2.3. Comparing this table to the data in Table 2.1 shows that the number of probes to the black box in our new algorithm does not depend on the degree of the target polynomial.

Benchmarks #3 and #4

These sets of problems consist of 14 random multivariate polynomials in $n = 6$ variables and $n = 12$ variables all of total degree $D = 30$. The i'th polynomial will have about 2^i

Table 2.2: benchmark #1: bad degree bound $d = 100$

i	t	New Algorithm		Zippel's Algorithm	
		Time	Probes	Time	Probes
1	2	0.00 (0.00)	13	0.01	707
2	4	0.00 (0.00)	25	0.01	1111
3	8	0.00 (0.00)	49	0.02	1818
4	16	0.00 (0.00)	97	0.03	2828
5	32	0.00 (0.00)	193	0.07	4949
6	64	0.01 (0.01)	385	0.14	8383
7	128	0.03 (0.01)	769	0.36	14342
8	253	0.09 (0.03)	1519	0.79	20907
9	512	0.29 (0.10)	3073	1.97	31714
10	1015	0.89 (0.31)	6091	3.97	40400
11	2041	3.08 (1.02)	12247	8.18	49288
12	4081	10.98 (3.61)	24487	15.16	52722
13	5430	18.92 (6.19)	32581	19.62	53530

non-zero terms. We run both the Zippel's algorithm and our new algorithm with degree bound $d = 30$. The timings and the number of probes are given in Tables 2.4 and 2.5.

Parallel benchmark.

To better assess the parallel implementation of our algorithm, Table 2.6 reports timings for benchmark #4 for an earlier version of our algorithm that we presented at PASCO (see [33]) in Grenoble, July 2010, running on 1, 2 and 4 cores of an Intel Core i7 processor running at 2.66GHz.

We report (in column roots) the time spent computing the roots in step 8 of $\Lambda_1(z)$ using an implementation of Rabin's algorithm which used classical polynomial arithmetic, and (in column solve) the time solving the linear system for the coefficients in step 13 and (in column probes) the total time spent probing the black box. The data shows that computing the roots will become a bottleneck for our parallel implementation for more cores. For $i = 13$ the sequential time is 484.6s. Of this, 34.7s was spent computing the roots of $\Lambda_1(z)$ and 5.02s was spent solving for the coefficients. Thus the algorithm has a sequential component of $34.7 + 5.02 = 39.7$s and so the maximum possible speedup on 4 cores is a factor of $484.6/((484.6 - 39.7)/4 + 39.7) = 3.21$ compared with the observed

Table 2.3: benchmark #2: $n = 3$ and $D = 100$

i	t	New Algorithm		Zippel		ProtoBox
		Time	Probes	Time	Probes	Probes
3	8	0.00 (0.00)	49	0.02	1919	89
4	16	0.00 (0.00)	97	0.04	3434	167
5	31	0.00 (0.00)	187	0.08	6161	320
6	64	0.01 (0.00)	385	0.19	10504	623
7	127	0.03 (0.01)	763	0.49	18887	1149
8	253	0.09 (0.03)	1519	1.38	32219	2137
9	511	0.29 (0.10)	3067	4.36	56863	4103
10	1017	0.91 (0.31)	6103	13.99	98677	7836
11	2037	3.07 (1.04)	12223	43.23	166650	DNF
12	4076	11.02 (3.61)	24457	121.68	262802	DNF
13	8147	40.68 (13.32)	48883	282.83	359863	DNF

speedup factor of $484.6/152.5 = 3.18$. On 12 cores the maximum speedup would be a poor $484.6/((484.6 - 39.7)/12 + 39.7) = 6.3$.

Table 2.7 reports timings for benchmark #4 for the our new algorithm with the asymptotically fast root finding algorithm running on two 6 core Intel Xeon X7460 CPUs running at 2.66GHz. We report timings and speedups for 4 and 12 cores. The data is extremely good showing a near linear speedup. For $i = 13$, if the solving were not parallelized, the maximum speedup for 12 cores would be, by Amdahl's law, $435.3/((435.3 - 4.2)/12 + 4.2) = 10.85$. However, by also parallelizing the coefficient solving step, (it is a simple observation that each coefficient can be solved for independently in $O(t)$ time) we obtain a speedup of 11.95 on 12 cores.

Benchmark #5

In this benchmark, we compare our new algorithm and the racing algorithm on seven target polynomials (below) from [40, p. 393]. Note, f_6 is dense. The number of probes for each algorithm is reported in Table 2.8.

$$f_1(x_1, \ldots, x_9) = x_1^2x_3^3x_4x_6x_8x_9^2 + x_1x_2x_3x_4^2x_5^2x_8x_9 +$$

$$x_2x_3x_4x_5^2x_8x_9 + x_1x_3^3x_4^2x_5^2x_6^2x_7x_8^2 + x_2x_3x_4x_5^2x_6x_7x_8^2$$

Table 2.4: benchmark #3: $n = 6$ and $D = 30$

i	t	New Algorithm		Zippel		ProtoBox
		Time	Probes	Time	Probes	Probes
3	8	0.00 (0.00)	97	0.01	1364	140
4	16	0.00 (0.00)	193	0.02	2511	284
5	31	0.00 (0.00)	373	0.05	4340	521
6	64	0.02 (0.01)	769	0.15	8060	995
7	127	0.06 (0.02)	1525	0.44	14601	1871
8	255	0.22 (0.07)	3061	1.51	27652	3615
9	511	0.72 (0.24)	6133	5.19	50530	6692
10	1016	2.43 (0.85)	12193	17.94	90985	12591
11	2037	8.69 (2.87)	24445	65.35	168299	DNF
12	4083	32.37 (10.6)	48997	230.60	301320	DNF
13	8151	122.5 (40.5)	97813	803.26	532549	DNF

$$f_2(x_1, \ldots, x_{10}) = x_1x_2^2x_4^2x_8x_9^2x_{10}^2 + x_2^2x_4x_5^2x_6x_7x_9x_{10}^2 +$$

$$x_1^2x_2x_3x_5^2x_7^2x_9^2 + x_1x_3^2x_4^2x_7^2x_9^2 + x_1^2x_3x_4x_7^2x_8^2$$

$$f_3(x_1, \ldots, x_9) = 9x_2^3x_3^3x_5^2x_6^2x_8^3x_9^3 + 9x_1^3x_2^2x_3^3x_5^2x_7^2x_8^2x_9^3 +$$

$$x_1^4x_3^4x_4^2x_5^4x_6^4x_7x_8^5x_9 + 10x_1^4x_2x_3^4x_4^4x_5^4x_7x_8^3x_9 + 12x_2^3x_4^3x_6^3x_7^2x_8^3$$

$$f_4(x_1, \ldots, x_9) = 9x_1^2x_3x_4x_6^3x_7^2x_8x_{10}^4 + 17x_1^3x_2x_5^2x_6^2x_7x_8^3x_9^4x_{10}^3 +$$

$$3x_1^3x_2^2x_6^3x_{10}^2 + 17x_2^2x_3^4x_4^2x_7^4x_8^3x_9x_{10}^3 + 10x_1x_3x_5^2x_6^2x_7^4x_8^4$$

$$f_5(x_1, \ldots, x_{50}) = \sum_{i=1}^{i=50} x_i^{50}$$

$$f_6(x_1, \ldots, x_5) = \sum_{i=1}^{i=5} (x_1 + x_2 + x_3 + x_4 + x_5)^i$$

$$f_7(x_1, x_2, x_3) = x_1^{20} + 2x_2 + 2x_2^2 + 2x_2^3 + 2x_2^4 + 3x_3^{20}$$

The reader may observe that in all benchmarks, the number of probes our algorithm makes is exactly $2nt + 1$.

Table 2.5: benchmark #4: $n = 12$ and $D = 30$

i	t	New Algorithm		Zippel		ProtoBox
		Time	Probes	Time	Probes	Probes
3	8	0.00 (0.00)	193	0.08	5053	250
4	15	0.00 (0.00)	361	0.20	10230	470
5	32	0.02 (0.01)	769	0.54	18879	962
6	63	0.06 (0.02)	1513	1.79	36735	1856
7	127	0.18 (0.05)	3049	6.10	69595	3647
8	255	0.62 (0.17)	6121	22.17	134664	7055
9	507	2.14 (0.55)	12169	83.44	259594	13440
10	1019	7.70 (1.94)	24457	316.23	498945	26077
11	2041	28.70 (7.23)	48985	1195.13	952351	DNF
12	4074	108.6 (27.2)	97777	4575.83	1841795	DNF
13	8139	421.1 (105.4)	195337	> 10000	-	DNF

2.5 Comparison of Different Algorithms

We have talked about some of the algorithms for sparse interpolation over finite fields. Table 2.9 compares these algorithms in terms of the number of probes, whether they are Las Vegas or Monte Carlo, whether they can be parallelized or not and the size of the prime. If the prime p is bigger that $(d+1)^n$, then the best algorithm to use is the Discrete Logs method [54] because it only does $O(t)$ probes to the black box. If the prime is not exponentially big, and we need a Las Vegas algorithm, one can use the method by Huang and Rao [29]. The problem with this method is the number of probes to the black box is significantly greater than the other algorithms. Both the racing algorithm by Kaltofen/Lee ([41, 40]) and our new algorithm do $O(nt)$ probes to the black box, but our algorithm can be parallelized easily. This is not the case for the racing algorithm. The algorithm by Garg and Schost [17] is for interpolating polynomials given by a straight-line program of size L. The key to their algorithm is that it works for *any* prime (no restriction on the size). Also the number of arithmetic operations that this algorithm does is logarithmic (and not linear) in the degree d.

In conclusion, our sparse interpolation algorithm is a modification of the Ben-Or/Tiwari algorithm [3] for polynomials over finite fields. It does a factor of $O(n)$ more probes where n is the number of variables. Our benchmarks show that for sparse polynomials, it does fewer

Table 2.6: Parallel speedup timing data for benchmark #4 for an earlier attempt.

		1 core				2 cores		4 cores	
i	t	time	roots	solve	probe	time	(speedup)	time	(speedup)
7	127	0.15	0.02	0.00	0.15	0.08	(1.87x)	0.06	(3x)
8	255	0.54	0.05	0.00	0.41	0.30	(1.80x)	0.18	(3x)
9	507	2.02	0.18	0.02	1.48	1.11	(1.82x)	0.67	(3.02x)
10	1019	7.94	0.65	0.08	5.76	4.35	(1.82x)	2.58	(3.08x)
11	2041	31.3	2.47	0.32	22.7	17.1	(1.83x)	9.94	(3.15x)
12	4074	122.3	9.24	1.26	90.0	67.1	(1.82x)	38.9	(3.14x)
13	8139	484.6	34.7	5.02	357.3	264.9	(1.82x)	152.5	(3.18x)

Table 2.7: Parallel speedup timing data for benchmark #4 for the new algorithm.

		1 core				4 cores		12 cores	
i	t	time	roots	solve	probe	time	(speedup)	time	(speedup)
7	127	0.218	0.01	0.00	0.12	0.062	(3.35x)	0.050	(4.2x)
8	255	0.688	0.01	0.01	0.40	0.186	(3.70x)	0.106	(6.5x)
9	507	2.33	0.05	0.02	1.53	0.603	(3.86x)	0.250	(9.3x)
10	1019	8.20	0.14	0.07	5.97	2.10	(3.90x)	0.748	(10.96x)
11	2041	30.17	0.34	0.26	23.6	7.62	(3.96x)	2.61	(11.56x)
12	4074	113.1	0.87	1.06	93.5	28.6	(3.96x)	9.90	(11.78x)
13	8139	435.3	2.25	4.20	371.7	110.5	(3.94x)	36.46	(11.95x)

probes to the black box than Zippel's algorithm and a comparable number to the racing algorithm of Kaltofen and Lee. Unlike Zippel's algorithm and the racing algorithm, our algorithm does not interpolate each variable sequentially and thus can easily be parallelized. Our parallel implementation using Cilk, demonstrates a very good speedup. The downside of our algorithm is that it is clearly worse than Zippel's algorithm and the racing algorithm for dense polynomials. This disadvantage is partly compensated for by the increased parallelism.

Although we presented our algorithm for interpolating over $\mathbb{Z}_p$, it also works over any finite field $GF(q)$. Furthermore, if p (or q) is too small, one can work inside a suitable extension field. We conclude with some remarks about the choice of p in applications where one may choose p.

Theorem 2.2 says that monomial collisions are likely when $\frac{dt^2}{2(p-1)} > \frac{1}{2}$, that is when $p - 1 < dt^2$. In our benchmarks we used 31 bit primes. Using such primes, if $d = 30$,

Table 2.8: benchmark #5.

i	n	d	$\#f_i$	New Algorithm	ProtoBox
1	9	3	5	90	126
2	10	2	5	100	124
3	9	3	5	90	133
4	9	4	5	100	133
5	50	50	50	5000	251
6	5	5	251	2510	881
7	3	20	6	36	41

Algorithm	# Probes	Deterministic?	Prime	Complexity
		Parallel?		
Ben-Or/Tiwari [3]	$2T$	Las Vegas	$p > p_n^d$	$O(T^2 + t^2 \log p + dt)$
		Yes		
Huang/Rao [29]	$O(dT^2)$	Las Vegas	$p > 8d^2t^2$	$> O((td)^{13})$
		Yes		
Discrete Logs 2006	$2T$	Las Vegas	$p > (d+1)^n$	$O(T^2 + t^2 \log p)$
		Yes		
Garg/Schost [17]	$O(T^2 \log d)$	Las Vegas	–	$O(T^4 \log^2 d)$
		Yes		
Zippel [84]	$O(ndt)$	Monte-Carlo	$p \gg nt$	$O(ndt^2)$
		Some		
Kaltofen/Lee [41]	$O(nt)$	Monte-Carlo	$p \gg ndt$	–
		Less		
Javadi/Monagan [33]	$2nT$	Monte-Carlo	$p \gg ndt^2$	$O(nT^2 + nt^2 \log p + ndt \log t)$
		Yes		

Table 2.9: Comparison of different algorithms.

monomial collisions will likely occur when $t > 8,460$ which means 31 bit primes are too small for applications where the number of terms t is large. The 31 bit prime limitation is a limitation of the C programming language. On a 64 bit machine, one can use 63 bit primes if one programs multiplication in $\mathbb{Z}_p$ in assembler. We are presently implementing this.

Chapter 3

GCD Computation

The problem of computing the *greatest common divisor* of two polynomials is one of the fundamental problems of computer algebra. It has many applications including simplifying rational expressions, polynomial factorization and symbolic integration. Having an efficient algorithm for computing GCDs is one of the most important parts of any general purpose computer algebra system (see [58] and Ch. 7 of [18]).

The most efficient way to solve the *coefficient growth problem* in the *Euclidean algorithm* is to use a *modular algorithm.* A modular algorithm projects the problem down to finding images of the GCD modulo a sequence of primes and then computing the final result using the Chinese remainder theorem. After computing the image of the GCD g, modulo a prime p, we use *trial division* to prove that the correct image of the GCD has been computed. The algorithms which use Zippel's sparse interpolation method [84, 85] will use the *form* of the GCD computed modulo the first prime (or evaluation point) to compute other images by constructing and solving systems of linear equations.

In [36], Kaltofen and Trager give an algorithm for computing the GCD of two multivariate polynomials over a field $\mathbb{F}$ of characteristic zero. They assume that the polynomials are represented with a black box. This algorithm along many other algorithms (e.g factorization, sparse interpolation,...) for polynomials represented with black boxes is implemented in the FOXBOX system (See [10]). FOXBOX is implemented in C++ on the top of SACLIB [27]. In principle one could use FOXBOX to find GCDs of multivariate polynomials over algebraic number fields by implementing components for these fields, though it is not optimized for these domains.

We will now give Zippel's modular GCD algorithm [84] for computing *monic* GCDs in $\mathbb{Z}[x_1, \dots, x_n]$. We have modified this algorithm for algebraic function fields and for non-monic inputs in [31].

Algorithm 3.1: Zippel's Modular GCD Algorithm - Subroutine **MGCD**

```
Require: $f_1, f_2 \in \mathbb{Z}[x_1, \dots, x_n]$ such that $g = \gcd(f_1, f_2)$ is monic [1] in the main variable
    $x_1$ and a degree bound $d$ on $\deg_{x_i}(g)$ for $1 \le i \le n$.
Ensure: Monic $g \in \mathbb{Z}[x_1, \dots, x_n]$.
 1: Choose a good [2] prime $p_1 \in \mathbb{Z}$ such that $p_1 \gg d$.
 2: Compute $g_1 = \gcd(f_1, f_2) \bmod p_1$ by calling Algorithm 3.2.
 3: Let $d_1 = \deg_{x_1}(g_p)$ and $g_1 = \sum_{i=0}^{d_1} C_i x_1^i$ where $C_i \in \mathbb{Z}_{p_1}[x_2, \dots, x_n]$ is the coefficient of
    $g_1$ in $x_1^i$.
 4: Let $C_i = \sum_{j=1}^{n_i} a_{ij} M_{ij}$ where $a_{ij} \in \mathbb{Z}$ and $M_{ij} \in \mathbb{Z}[x_2, \dots, x_n]$ is a monomial of $C_i$.
    Let $g_f = \sum_{i=0}^{d_1} \bar{C}_i x_1^i$ where $\bar{C}_i = \sum_{j=1}^{n_i} A_{ij} M_{ij}$ is the form for $C_i$ with unknown integer
    coefficients $A_{ij}$s. Here $g_f$ is the form of the GCD.
 5: repeat
 6:     Choose a new good prime $p_l \in \mathbb{Z}$.
 7:     Compute the $g_l = \gcd(f_1, f_2) \bmod p_l$ using Algorithm 3.3 on inputs $f_1 \bmod p_l$, $f_2 \bmod$
        $p_l$ and $g_f$ over $\mathbb{Z}_{p_l}$.
 8:     If $g_l$ is FAIL, then restart the algorithm.
 9:     If $g_l$ is UNLUCKY then go to Step 6.
10:     Use the Chinese remaindering algorithm to compute $g \in \mathbb{Z}_{\prod p_i}[x_1, \dots, x_n]$, such that
        $g \equiv g_i \bmod p_i$ for $1 \le i \le l$.
11: until $g \mid f_1$ and $g \mid f_2$ over $\mathbb{Z}$.
12: return $g$.
```

Algorithm 3.2: Zippel's Modular GCD Algorithm - Subroutine **PGCD**

```
Require: $f_1, f_2 \in \mathbb{Z}_p[x_1, \dots, x_n]$ such that $g = \gcd(f_1, f_2) \bmod p$ is monic in the main
    variable $x_1$. A degree bound $d$ on $\deg_{x_i}(g)$ for $1 \le i \le n$.
```

[1] In fact the GCD could be of the form $g = a x_1^{d_1} + h(x_1, \dots, x_n)$ where $a \in \mathbb{Z}$ and $\deg_{x_1}(h) < d_1$.

[2] The primes must not divide the leading coefficient of any of the inputs in the main variable x_1.

```
Ensure: Monic $g \in \mathbb{Z}_p[x_1, \ldots, x_n]$.
1: If $n = 1$ return the output of the Euclidean algorithm on inputs $f_1, f_2 \in \mathbb{Z}_p[x_1]$ .
2: Choose a good [3] evaluation point $\alpha_1 \in \mathbb{Z}_p$ at random. Let $a = f_1(x_1, \ldots, x_{n-1}, x_n = \alpha_1) \bmod p$ and $b = f_2(x_1, \ldots, x_{n-1}, x_n = \alpha_1) \bmod p$. We have $a, b \in \mathbb{Z}_p[x_1, \ldots, x_{n-1}]$.
3: Apply Algorithm 3.2 recursively to compute $g_1 = \gcd(a, b) \bmod p$.
4: Let $g_f$ be the form of the GCD obtained from $g_1$.
5: for $i$ from 2 to $d + 1$ do
6:     Choose a good evaluation point $\alpha_i \in \mathbb{Z}_p$ at random. Let $a = f_1(x_1, \ldots, x_{n-1}, x_n = \alpha_i) \bmod p$ and $b = f_2(x_1, \ldots, x_{n-1}, x_n = \alpha_i) \bmod p$.
7:     Let $g_i$ be the output of Algorithm 3.3 on $a, b \in \mathbb{Z}_p[x_1, \ldots, x_{n-1}]$ and $g_f$ over $\mathbb{Z}_p$.
8:     If $g_i$ is FAIL then restart the algorithm.
9:     If $g_i$ is UNLUCKY then go to Step 6.
10: end for
11: Use Newton's interpolation algorithm (dense) to interpolate $x_n$ in $g$ such that $g \equiv g_i \bmod \langle x_n - \alpha_i \rangle$ for $1 \le i \le d + 1$ and $\deg_{x_i}(g) \le d$.
12: If $g \mid f_1 \bmod p$ and $g \mid f_2 \bmod p$ then return $g$ otherwise restart the algorithm.
```

Algorithm 3.3: **Zippel's Sparse Interpolation Algorithm**

```
Require: $f_1, f_2 \in R[x_1, \ldots, x_n]$ such that $g = \gcd(f_1, f_2)$ is monic in the main variable $x_1$. $R$ here is the coefficient ring (either $\mathbb{Z}$ or $\mathbb{Z}_p$ for some prime $p$). And $g_f$, the form of the GCD $g$ which is of degree $d_1$ in $x_1$.
Ensure: Either fail (UNLUCKY or FAIL) or the monic GCD $g \in R[x_1, \ldots, x_n]$.
1: Let $g_f = \sum_{i=0}^{d_1} \bar{C}_i x_1^i$ where $\bar{C}_i = \sum_{j=1}^{n_i} A_{ij} M_{ij}$ is the form for $C_i$, the coefficient of $g$ in $x_1^i$. Here the coefficient $A_{ij} \in R$ is unknown and $M_{ij}$ does not have $x_1$.
2: Let $N = \max(n_0, \ldots, n_{d_1}) + 1$ [4] where $n_i$ is the number of non-zero terms in $C_i$.
3: Choose $N$ random evaluation points $\beta_k = (\alpha_1, \ldots, \alpha_{n-1}) \in R^{n-1}$ $(1 \le k \le N)$.
4: Let $a_k = f_1(x_1, \beta_k)$ and $b_k = f_2(x_1, \beta_k)$. We have $a_k, b_k \in R[x_1]$. Use the Euclidean algorithm to compute $\bar{g}_k = \gcd(a_k, b_k) \in R[x_1]$ for $1 \le k \le N$ s.t. $g_k$ is monic.
5: If for some $1 \le k \le N$, $\deg_{x_1}(\bar{g}_k) < d_1$ then return FAIL. {Recall that $d_1 = \deg_{x_1}(g_f)$.}
6: If for some $1 \le k \le N$, $\deg_{x_1}(\bar{g}_k) > d_1$ then choose a new evaluation point $\beta_k$ and
```

[3] The evaluation point α is good if $\mathrm{lc}_{x_1}(f_1)(\alpha) \neq 0$ and $\mathrm{lc}_{x_1}(f_2)(\alpha) \neq 0$.

[4] The one extra evaluation point is to possibly detect any error in the form g_f.

```
   compute $\bar{g}_k = \gcd(a_k, b_k) \in R[x_1]$. Repeat this until $\deg_{x_1}(\bar{g}_k) = d_1$ for $1 \le k \le N$.
7: For $0 \le i \le d_1$, let $S_i$ be a linear system obtained by equating $\sum_{j=1}^{n_i} m_{ij}^k A_{ij}$ to $c_i^k$ where
   for $1 \le k \le N$, $c_i^k$ is the coefficient of $g_k$ in $x_1^i$ and $m_{ij}^k = M_{ij}(\beta_k) \in R$.
8: Solve the linear system of equations $S_i$ for $1 \le i \le d_1$. If $S_i$ is under-determined then
   return UNLUCKY. If $S_i$ is inconsistent, then return FAIL. If the system is determined
   and consistent, we have the values for the unknown $A_{ij}$s. Substitute these values in $g_f$
   to obtain $g \in R[x_1, \ldots, x_n]$.
9: return $g$.
```

Zippel's sparse interpolation assumed that if an image of a polynomial obtained from a random evaluation point is zero, then the polynomial itself is zero *with high probability.* Schwartz (Lemma 1.7) states that for $\alpha_1, \ldots, \alpha_n \in \mathbb{Z}_p$, we would have $f(\alpha_1, \ldots, \alpha_n) = 0$ for non-zero f with probability at most $\frac{d}{p}$ where $d = \deg(f)$. In Algorithm 3.2, let α_1 be the first evaluation point. Let $g = \gcd(f_1, f_2) \in (\mathbb{Z}_p[x_n])[x_1, \ldots, x_{n-1}]$ and $g = \sum_{i=1}^{T} C_i M_i$ where $C_i \in \mathbb{Z}_p[x_n]$ and M_i is a monomial in $x_1, \ldots, x_{n-1}$. The assumption in Zippel's algorithm is that if $C_i(\alpha_1) \equiv 0 \bmod p$ then $C_i \equiv 0 \bmod p$ with high probability [5].

One of the bottlenecks of Zippel's modular GCD algorithm is the trial divisions which are done to ensure that the GCD computed is correct (Step 11 of Algorithm 3.1 and Step 12 of Algorithm 3.2). These trial divisions are often expensive, especially if the GCD is dense.

The SparseModGcd Algorithm

Our main goal in this chapter is improving the SparseModGcd algorithm we presented in [30, 31]. Let $F = \mathbb{Q}(t_1, \ldots, t_k)$. For i, $1 \le i \le r$, let $m_i(z_1, \ldots, z_i) \in F[z_1, \ldots, z_i]$ be monic and irreducible over $F[z_1, \ldots, z_{i-1}]/\langle m_1, \ldots, m_{i-1}\rangle$. Let $L = F[z_1, \ldots, z_r]\ /\langle m_1, \ldots, m_r\rangle$. L is an algebraic function field in k parameters $t_1, \ldots, t_k$. Suppose f_1 and f_2 are non-zero polynomials in $L[x_1, \ldots, x_n]$. The SparseModGcd algorithm will compute g or an associate (scalar multiple) of g. The algorithm is modular and uses Zippel's sparse interpolation algorithm. We also use rational function reconstruction to recover the coefficients of the GCD, when interpolating a parameter or a variable. Before getting into the details of the improvements we need to give some definitions.

Definition 3.1. Let $D = \mathbb{Z}[t_1, \ldots, t_k]$. A non-zero polynomial in $D[z_1, \ldots, z_r, x]$ is said to be *primitive* with respect to $(z_1, \ldots, z_r, x)$ if the GCD of its coefficients in D is 1. Let f be

[5] According to Lemma 1.7, if $C_i(\alpha) \equiv 0 \bmod p$ than $\text{Prob}(C_i \ne 0 \bmod p) \le \frac{\deg(C_i(x_2,\ldots,x_n))}{p}$

non-zero in $L[x]$ where L is the algebraic function field previously defined. The denominator of f is the polynomial $den(f) \in D$ of least total degree in $(t_1, \ldots, t_k)$ and with smallest integer content such that $den(f)f$ is in $D[z_1, \ldots, z_r, x]$. The primitive associate $\check{f}$ of f is the associate of $den(f)f$ which is primitive in $D[z_1, \ldots, z_r, x]$ and has positive leading coefficient in a term ordering.

The following is an example from [31].

Example 3.2. Let $f = 3tx^2 + 6tx/(t^2 - 1) + 30tz/(1 - t)$ where $m_1(z) = z^2 - t$. Here $f \in L[x]$ where $L = \mathbb{Q}(t)[z]/\left\langle z^2 - t \right\rangle$ is an algebraic function field in one parameter t. We have $\mathrm{den}(f) = t^2 - 1$ and $\check{f} = \mathrm{den}(f)f/(3t) = (t^2 - 1)x + 2x - 10z(t + 1)$.

Definition 3.3 (See [9])**.** Recall that for a polynomial $f = a_n x^n + \cdots + a_1 x + a_0$ where $f \in \mathbb{F}[x]$ the *content* of f in x is

$$\mathrm{cont}_x(f) = \gcd(a_0, a_1, \ldots, a_n) \in \mathbb{F}.$$

A prime p, is said to introduce an *unlucky content* if for two input polynomials $f_1, f_2 \in \mathbb{F}[x_1, \ldots, x_n]$ with GCD $g = \gcd(f_1, f_2)$, $\mathrm{cont}_{x_1}(g) = 1$ but $\mathrm{cont}_{x_1}(g \bmod p) \neq 1$. Similarly an evaluation point $x_i = \alpha_j \in \mathbb{F}$ is said to introduce an unlucky content if $\mathrm{cont}_{x_1}(g) = 1$ but $\mathrm{cont}_{x_1}(g(x_j = \alpha_j)) \neq 1$.

Here is an example of an unlucky content from [31].

Example 3.4. Suppose $g = (12s+t)x+(s+12t)z$. We have $\mathrm{cont}_x(g) = \gcd(12s+t, s+12t) = 1$. But for $p = 11$ we obtain $\mathrm{cont}_x(g \bmod p) = s + t$. Hence $p = 11$ introduces an unlucky content and, for any prime p (other than 2 and 3), the evaluation points $t = 0$ and $s = 0$ introduce unlucky contents.

Suppose during sparse interpolation we choose our assumed form, g_f, based on an image which is computed modulo a prime (or evaluation point) which introduced an unlucky content, e.g. $p_1 = 11$ in our example. Then the assumed form g_f will have different terms in $x, z_1, \ldots, z_r, t_1, \ldots, t_k$ than in g. This will with high probability result in an inconsistent linear system during sparse interpolation.

Suppose instead that our assumed form g_f is correct but it is a subsequent prime or evaluation point that introduces an unlucky content. This *may* lead to an under-determined linear system.

Example 3.5. Consider $f_1 = f_2 = zx + t + 1$ where $m(z) = z^2 - t - 14$. Here $g = x + (t+1)/(t+14)z$ and hence $\breve{g} = (t+14)x + (t+1)z$ thus $p = 13$ introduces an unlucky content $t+1$. Suppose our first prime is $p_1 \neq 13$ and we obtain the correct assumed form $g_f = (At+B)x + (Ct+D)z$. Suppose our second prime is $p_2 = 13$ and we perform a sparse interpolation in t using $t = 1, 2, 3, \ldots$. Since g_f is not monic in x we will equate $g_f(t) = m_t\breve{g}(t)$ and solve for $A, B, C, D, m_1, m_2, \ldots$ with $m_1 = 1$. For $t = 1, 2, 3$ we obtain the following equations modulo 13.

$$\begin{aligned}(A+B)x + (C+D)z &= m_1(x+z),\\ (2A+B)x + (2C+D)z &= m_2(x+z),\\ (3A+B)x + (3C+D)z &= m_3(x+z).\end{aligned}$$

Equating coefficients of x^iz^j we obtain the following linear system: $A+B = m_1$, $2A+B = m_2$, $3A+B = m_3$, $C+D = 1$, $2C+D = 1$, $3C+D = 1$. The reader may verify that this system, with $m_1 = 1$, is not determined, and also, adding further equations, for example, from $t = 4$, does not make the system determined.

If the system of linear equations is not determined (with one more image than necessary), SparseModGcd will assume that the current prime or one of the evaluations has caused an unlucky content. In this case the algorithm restarts with a new prime (Similar to Step 8 of Algorithm 3.3).

The main bottleneck of the SparseModGcd algorithm presented in [31] on sparse input polynomials is computing the univariate GCDs of polynomials over an algebraic number ring modulo a prime p. In Section 3.1 we will discuss how we can improve this by implementing an *in-place* GCD algorithm. In Section 3.2 we will prove that the probability of algorithm MQRR making an error is low. We will also give the probability that Zippel's sparse interpolation will succeed, given a wrong form (due to the error in the output of MQRR). Using these we prove that we can eliminate the trial divisions in positive characteristic. In Section 3.3 we will show that the previous solution given for *normalization problem* in [9] has an error. We will give a new solution for this problem and prove that this solution works.

3.1 Univariate GCDs over Algebraic Number Fields

In 2002, van Hoeij and Monagan in [70] presented an algorithm for computing the monic GCD $g(x)$ of two polynomials $f_1(x)$ and $f_2(x)$ in $L[x]$ where $L = \mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_k)$ is an algebraic number field. The algorithm is a *modular* GCD algorithm. It computes the GCD of f_1 and f_2 modulo a sequence of primes $p_1, p_2, \ldots, p_l$ using the monic Euclidean algorithm in $L_p[x]$ ($L_p = L \bmod p$) and it reconstructs the rational numbers in $g(x)$ using Chinese remaindering and rational number reconstruction. The algorithm is a generalization of earlier work of Langymyr and MaCallum [48], and Encarnación [12] to treat the case where L has multiple extensions ($k > 1$). It can be generalized to multivariate polynomials in $L[x_1, x_2, \ldots, x_n]$ using evaluation and interpolation (see [71, 31]).

Monagan implemented the algorithm in Maple in 2001 and in Magma in 2003 using the *recursive dense* polynomial representation to represent elements of L, L_p, $L[x_1, \ldots, x_n]$ and $L_p[x_1, \ldots, x_n]$. This representation is generally more efficient than the distributed and recursive sparse representations for sparse polynomials. See for example the comparison by Fateman in [14]. And since efficiency in the recursive dense representation improves for dense polynomials, and elements of L are often dense, it should be a good choice for implementing arithmetic in L and also L_p.

However, we have observed that arithmetic in L_p is very slow when α_1 has low degree. Since this case often occurs in practical applications, and since over 90% of a GCD computation in $L[x]$ is typically spent in the Euclidean algorithm in $L_p[x]$, we sought to improve the efficiency of the arithmetic in L_p. One reason why this happens is because the cost of storage management, allocating small arrays for storing intermediate polynomials of low degree can be much higher than the cost of the actual arithmetic being done in $\mathbb{Z}_p$.

We explain why this is the case with an example.

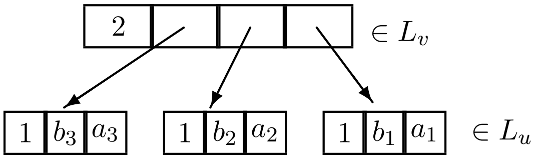
Example 3.6. Let $L = \mathbb{Q}(\alpha_1,\ \alpha_2)$ where $\alpha_1 = \sqrt{2}$ and $\alpha_2 = \sqrt[3]{1/5 + \alpha_1}$. L is an algebraic number field of degree $d = 6$ over $\mathbb{Q}$. We represent elements of L as polynomials in $\mathbb{Q}[u][v]$ and we do arithmetic in L modulo the ideal $I = \langle m_1(u), m_2(v, u)\rangle$ where $m_1(u) = u^2 - 2$ and $m_2(v, u) = v^3 - u - 1/5$ are the minimal polynomials for α_1 and, respectively, α_2.

To implement the modular GCD algorithm one uses *machine primes*, that is, the largest available primes that fit in the word of the computer so that arithmetic in $\mathbb{Z}_p$ can be done by the computer's hardware. After choosing the next machine prime p, we build the ring $L_p[x]$ where $L_p = L \bmod p$, iteratively, as follows; first we build the residue ring

$L_u = \mathbb{Z}_p[u]/\langle u^2 - 2 \bmod p\rangle$. We use a dense array of machine integers to represent elements of L_u. Then we build $L_v = L_u[v]/\langle v^3 - u - 1/5 \bmod p\rangle$ and finally the polynomial ring $L_p[x]$. In the recursive dense representation we represent elements of L_v as dense arrays of pointers to elements of L_u. So a general element of L_v, which looks like

$$(a_1u + b_1)v^2 + (a_2u + b_2)v + (a_3u + b_3),$$

would be stored as follows where the degree of each element is explicitly stored.



When the monic Euclidean algorithm is executed in $L_p[x]$, it will do many multiplications and additions of elements in L_v, each of which will do many in L_u. This results in many calls to the storage manager to allocate small arrays for intermediate and final results in L_u and L_v and rapidly produces a lot of small pieces of garbage. Consider one such multiplication in L_u

$$(au + b)(cu + d) \bmod u^2 - 2.$$

The algorithms compute the product $P = acu^2 + (ad + bc)u + bd$ and then divide P by $u^2 - 2$ to get the remainder $R = (ad + bc)u + (bd + 2ac)$. They allocate arrays to store the polynomials P and R. We have observed that, even though the storage manager is not inefficient, the cost of these storage allocations and the other overhead for arithmetic in $\mathbb{Z}_p[u]/\langle u^2 - 2\rangle$ overwhelms the cost of the actual integer arithmetic in $\mathbb{Z}_p$ needed to compute $(ad + bc) \bmod p$ and $(bd + 2ac) \bmod p$.

Our main contribution is a library of *in-place* [6] algorithms for arithmetic in L_p and $L_p[x]$ where L_p has one or more extensions. The main idea is to eliminate all calls to the storage manager by pre-allocating one large piece of working storage, and re-using parts of it in a computation. In Section 3.1.1 we describe the recursive dense polynomial representation for

[6] In-place here means the total amount of storage required is pre-computed so that only one call to the storage allocator is necessary. This storage is used many times for intermediate results. The term *in-place* is also used for algorithms which overwrite their input buffer with the output (See e.g. [25]).

elements of $L_p[x]$. In Section 3.1.2 we present algorithms for multiplication and inversion in L_p and multiplication, division with remainder and GCD in $L_p[x]$ which are given one array of storage in which to write the output and one additional array W of working storage for intermediate results. In Section 3.1.3 we give formula for determining the size of W needed for each algorithm. In each case the amount of working storage is linear in d the degree of L. We have implemented our algorithms in the C language in a library which includes also algorithms for addition, subtraction, and other utility routines. In Section 3.1.4 we present benchmarks demonstrating its efficiency by comparing our algorithms with the Magma ([4]) computer algebra system and we explain how to avoid most of the integer divisions by p when doing arithmetic in $\mathbb{Z}_p$ because this also significantly affects overall performance.

3.1.1 Polynomial Representation

Let $\mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_r)$ be our number field L. We build L as follows. For $1 \leq i \leq r$, let $m_i(z_1, \ldots, z_i) \in \mathbb{Q}[z_1, \ldots, z_i]$ be the minimal polynomial for α_i, monic and irreducible over $\mathbb{Q}[z_1, \ldots, z_{i-1}]/\langle m_1, \ldots, m_{i-1}\rangle$. Let $d_i = \deg_{z_i}(m_i)$. We assume $d_i \geq 2$. Let $L = \mathbb{Q}[z_1, \ldots, z_r]/\langle m_1, \ldots, m_r\rangle$. So L is an algebraic number field of degree $d = \prod d_i$ over $\mathbb{Q}$. For a prime p for which the rational coefficients of m_i exist modulo p, let $R_i = \mathbb{Z}_p[z_1, \ldots, z_i]/\langle \bar{m}_1, \ldots, \bar{m}_i\rangle$ where $\bar{m}_i = m_i \bmod p$ and let $R = R_r = L \bmod p$. We use the following recursive dense representation for elements of R and polynomials in $R[x]$ for our algorithms. We view an element of R_{i+1} as a polynomial with degree at most $d_{i+1} - 1$ with coefficients in R_i.

To represent a non-zero element $\beta_1 = a_0 + a_1 z_1 + \cdots + a_{d_1-1} z_1^{d_1-1} \in R_1$ we use an array A_1 of size $S_1 = d_1 + 1$ indexed from 0 to d_1, of integers (modulo p) to store β_1. We store $A_1[0] = \deg_{z_1}(\alpha_1)$ and, for $0 \leq i < d_1 : A_1[i+1] = a_i$. Note that if $\deg_{z_1}(\alpha_1) = \bar{d} < d_1 - 1$ then for $\bar{d} + 1 < j \leq d_1$, $A_1[j] = 0$. To represent the zero element of R_1 we use $A[0] = -1$.

Now suppose we want to represent an element $\beta_2 = b_0 + b_1 z_2 + \cdots + b_{d_2-1} z_2^{d_2-1} \in R_2$ where $b_i \in R_1$ using an array A_2 of size $S_2 = d_2 S_1 + 1 = d_2(d_1+1)+1$. We store $A_2[0] = \deg_{z_2}(\beta_2)$ and for $0 \leq i < d_2$

$$A_2[i(d_1+1)+1 \ldots (i+1)(d_1+1)] = B_i[0 \ldots d_1],$$

where B_i is the array which represents $b_i \in R_1$. Again if $\beta_2 = 0$ we store $A_2[0] = -1$.

Similarly, we recursively represent $\beta_r = c_0 + c_1 z_r + \cdots + c_{d_r-1} z_r^{d_r-1} \in R_r$ based on the representation of $c_i \in R_{r-1}$. Let $S_r = d_r S_{r-1} + 1$ and suppose A_r is an array of size S_r such that $A_r[0] = \deg_{z_r}(\beta_r)$ and for $0 \le i < d_r$

$$A_r[i(d_{r-1}) + 1 \ldots (i+1)(d_{r-1}+1)] = C_i[0 \ldots S_{r-1} - 1].$$

Note, we store the degrees of the elements of R_i in $A_i[0]$ simply to avoid re-computing them. We have

$$\prod_{i=1}^{r} d_i < S_r < \prod_{i=1}^{r} (d_i + 1), S_r \in O(\prod_{i=1}^{r} d_i).$$

Now suppose we use the array C to represent a polynomial $f \in R_i[x]$ of degree d_x in the same way. Each coefficient of f in x is an element of R_i which needs an array of size S_i, hence C must be of size

$$P(d_x, R_i) = (d_x + 1)S_i + 1.$$

Example 3.7. Let $r = 2$ and $p = 17$. Let $\bar{m}_1 = z_1^3 + 3$, $\bar{m}_2 = z_2^2 + 5z_1z_2 + 4z_2 + 7z_1^2 + 3z_1 + 6$, and $f = 3 + 4z_1 + (5 + 6z_1)z_2 + (7 + 8z_1 + 9z_1^2 + (10z_1 + 11z_1^2)z_2)x + 12x^2$.
The representation for f is

$$C = \begin{array}{|c|}\hline 2\\\hline\end{array}\underbrace{\begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 1&1&3&4&0&1&5&6&0\\\hline\end{array}}_{3+4z_1+(5+6z_1)z_2}\begin{array}{|c|c|c|c|c|}\hline 1&2&7&8&9\\\hline\end{array}\underbrace{\begin{array}{|c|c|c|c|}\hline 2&0&10&11\\\hline\end{array}}_{10z_1+11z_1^2}\begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 0&0&12&0&0&-1&0&0&0\\\hline\end{array}$$

Here $d_x = 2, d_1 = 3, d_2 = 2, S_1 = d_1 + 1 = 4, S_2 = d_2S_1 + 1 = 9$ and the size of the array A is $P(d_x, R_2) = (d_x + 1)S_2 + 1 = 28$.

We also need to represent the minimal polynomial $\bar{m}_i$. Let $\bar{m}_i = a_0 + a_1z_i + \ldots a_{d_i}z_i^{d_i}$ where $a_j \in R_{i-1}$. We need an array of size S_{i-1} to represent a_j so to represent $\bar{m}_i$ in the same way we described above, we need an array of size $\bar{S}_i = 1 + (d_i + 1)S_{i-1} = d_iS_{i-1} + 1 + S_{i-1} = S_i + S_{i-1}$. We define $S_0 = 1$.

We represent minimal polynomials $\{\bar{m}_1, \ldots, \bar{m}_r\}$ as an array E, of size $\sum_{i=1}^{r} \bar{S}_i = \sum_{i=1}^{r} (S_i + S_{i-1}) = 1 + S_r + 2\sum_{i=1}^{r-1} S_i$ such that $E[M_i \ldots M_{i+1} - 1]$ represents m_{r-i} where $M_0 = 0$ and $M_i = \sum_{i=r-i+1}^{r} \bar{S}_i$. The minimal polynomials in Example 3.7 will be represented in the following figure where $E[0 \ldots 12]$ represents $\bar{m}_2$ and $E[13 \ldots 17]$ represents $\bar{m}_1$.

$$E = \underbrace{\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 2&2&6&3&7&1&4&5&0&0&1&0&0\\\hline\end{array}}_{\bar{m}_2}\underbrace{\begin{array}{|c|c|c|c|c|}\hline 3&3&0&0&1\\\hline\end{array}}_{\bar{m}_1}$$

3.1.2 In-place Algorithms

In this section we design efficient in-place algorithms for multiplication, division and GCD computation of two univariate polynomials over R. We will also give an in-place algorithm for computing the inverse of an element $\alpha \in R$, if it exists. This is needed for making a polynomial monic for the monic Euclidean algorithm in $R[x]$. We assume the following utility operations are implemented.

- IP_ADD(N, A, B) and IP_SUB(N, A, B) are used for in-place addition and subtraction of two polynomials $a, b \in R_N[x]$ represented in arrays A and B.
- IP_MUL_NO_EXT is used for multiplication of two polynomials over $\mathbb{Z}_p$. A description of this algorithm is given in Section 3.1.4.
- IP_REM_NO_EXT is used for computing the quotient and the remainder of dividing two polynomials over $\mathbb{Z}_p$.
- IP_INV_NO_EXT is used for computing the inverse of an element in $\mathbb{Z}_p[z]$ modulo a minimal polynomial $m \in \mathbb{Z}_p[z]$.
- IP_GCD_NO_EXT is used for computing the GCD of two univariate polynomials over $\mathbb{Z}_p$ (the Euclidean algorithm, See [58]).

In-place Multiplication

Suppose we have $a, b \in R[x]$ where $R = R_{r-1}[z_r]/\langle m_r(z_r)\rangle$. Let $a = \sum_{i=0}^{d_a} a_i x^i$ and $b = \sum_{i=0}^{d_b} b_i x^i$ where $d_a = \deg_x(a)$ and $d_b = \deg_x(b)$ and Let $c = a \times b = \sum_{i=0}^{d_c} c_i x^i$ where $d_c = \deg_x(c) = d_a + d_b$. To reduce the number of divisions by $m_r(z_r)$ when multiplying $a \times b$, we use the Cauchy product rule to compute c_k as suggested in [58], that is,

$$c_k = \left[\sum_{i=\max(0,k-d_b)}^{\min(k,d_a)} a_i \times b_{k-i} \right] \mod m_r(z_r).$$

Thus the number of multiplications in $R_{r-1}[z_r]$ (in line 11) is $(d_a + 1) \times (d_b + 1)$ and the number of divisions in $R_{r-1}[z_r]$ (in line 15) is $d_a + d_b + 1$.

The number of iterations in the loop in line 10 is $K = \min(k, d_a) - \max(0, k - d_b) + 1$. We have $0 \le k \le d_c = d_a + d_b$ thus

$$1 \le K \le \min(d_a, d_b) + 1.$$

Here instead of doing K divisions in $R_{r-1}[z_r]$, we do only one. This saves about half the work.

Algorithm 3.4: Algorithm IP_MUL: In-place Multiplication

```
Require:
  • $N$ the number of field extensions.
  • Arrays $A[0\dots\bar{a}]$ and $B[0\dots\bar{b}]$ representing univariate polynomials $a, b \in R_N[x]$ ($R_N = \mathbb{Z}_p[z_1,\dots,z_N]/\langle \bar{m}_1,\dots,\bar{m}_N\rangle$). Note that $\bar{a} = P(d_a, R_N) - 1$ and $\bar{b} = P(d_b, R_N) - 1$ where $d_a = \deg_x(a)$ and $d_b = \deg_x(b)$.
  • Array $C[0\dots\bar{c}]$: Space needed for storing $c = a \times b = \sum_{i=0}^{d_c} c_i x^i$ where $\bar{c} = P(\deg_x(a) + \deg_x(b), R_N) - 1$.
  • $E[0\dots e_N]$ : representing the set of minimal polynomials where $e_N = S_N + 2\sum_{i=1}^{N-1} S_i$.
  • $W[0\dots w_N]$ : *the working storage* for the intermediate operations.
Ensure: For $0 \le k \le d_c$, $c_k$ will be computed and stored in $C[kS_N + 1\dots(k+1)S_N]$.
Set $d_a := A[0]$ and $d_b := B[0]$.
if $d_a = -1$ or $d_b = -1$ then Set $C[0] := -1$ and return.
if $N = 0$ then Call IP_MUL_NO_EXT on inputs $A$, $B$ and $C$ and return.
Let $M = E[0\dots\bar{S}_N - 1]$ and $E' = E[\bar{S}_N\dots e_N]$ {$M$ points to $\bar{m}_N$ in $E[0\dots e_N]$}.
Let $T_1 = W[0\dots t-1]$ and $T_2 = W[t\dots 2t-1]$ and $W' = W[2t\dots w_N]$ where $t = P(2d_N - 2, R_{N-1})$ and $d_N = M[0] = \deg_{z_N}(\bar{m}_N)$.
Set $d_c := d_a + d_b$ and $s_c := 1$.
for $k$ from 0 to $d_c$ do
    Set $s_a := 1 + iS_N$ and $s_b := 1 + (k-i)S_N$.
    Set $T_1[0] := -1$ {$T_1 = 0$}.
    for $i$ from $\max(0, k - d_b)$ to $\min(k, d_a)$ do
        Call IP_MUL($N-1$, $A[s_a\dots\bar{a}]$, $B[s_b\dots\bar{b}], T_2, E', W'$).
        Call IP_ADD($N-1, T_1, T_2$) {$T_1 := T_1 + T_2$}
        Set $s_a := s_a + S_N$ and $s_b := s_b - S_N$.
    end for
    Call IP_REM($N-1, T_1, M, E', W'$). {Reduce $T_1$ modulo $M = \bar{m}_N$}.
    Copy $T_1[0\dots S_N - 1]$ into $C[sc\dots sc + S_N - 1]$.
    Set $s_c := s_c + S_N$.
end for
```

```
19: Determine $\deg_x(a \times b)$: {There might be zero-divisors}.
20: Set $i := d_c$ and $s_c := s_c - S_N$.
21: **while** $i \geq 0$ and $C[sc] = -1$ **do** Set $i := i - 1$ and $s_c := s_c - S_N$.
22: Set $C[0] := i$.
```

The temporary variables T_1 and T_2 must be big enough to store the product of two coefficients in $a, b \in R_N[x]$. Coefficients of a and b are in $R_{N-1}[z_N]$ with degree (in z_N) at most $d_N - 1$. Hence these temporaries must be of size $P(d_N - 1 + d_N - 1, R_{N-1}) = P(2d_N - 2, R_{N-1})$.

In-place Division

The following algorithm divides a polynomial $a \in R_N[x]$ by a *monic* polynomial $b \in R_N[x]$. The remainder and the quotient of a divided by b will be stored in the array representing a hence a is destroyed by the algorithm. The division algorithm is organized differently from the normal long division algorithm which does $d_b \times (d_a - d_b + 1)$ multiplications and divisions in $R_{N-1}[z_r]$. The total number of divisions by M in $R_{N-1}[z_r]$ in line 16 is reduced to $d_a + 1$ (see line 8).

The number of iterations in the loop in line 11 is $K = \min(D_r, k) - \max(0, k - D_q) + 1$. We have $0 \leq k \leq d_a$ thus

$$1 \leq K \leq \min(D_r, D_q) + 1 = \min(d_b, d_a - d_b + 1).$$

Here instead of doing K divisions in $R_{N-1}[z_r]$, we do only one. This saves about half the work.

Algorithm 3.5: Algorithm IP_REM: In-place Remainder

Require:

- N the number of field extensions.
- Arrays $A[0 \ldots \bar{a}]$ and $B[0 \ldots \bar{b}]$ representing univariate polynomials $a, b \neq 0 \in R_N[x]$ $(R_N = \mathbb{Z}_p[z_1, \ldots, z_N]/\langle \bar{m}_1, \ldots, \bar{m}_N \rangle)$ where $d_a = \deg_x(a) \geq d_a = \deg_x(b)$. Note b must be monic and $\bar{a} = P(d_a, R_N) - 1$ and $\bar{b} = P(d_b, R_N) - 1$.
- $E[0 \ldots e_N]$: representing the set of minimal polynomials where $e_N = S_N + 2\sum_{i=1}^{N-1} S_i$.

```
    • $W[0 \ldots w_N]$ : *the working storage* for the intermediate operations.
Ensure: The remainder $\bar{R}$ of $a$ divided by $b$ will be stored in $A[0 \ldots \bar{r}]$ where $\bar{r} = P(D, R_N) - 1$ and $D = \deg_x(\bar{R}) \leq d_b - 1$. Also let $Q$ represent the quotient $\bar{Q}$ of $a$ divided by $b$. $Q[1 \ldots \bar{q}]$ will be stored in $A[1 + d_b S_N \ldots \bar{a}]$ where $\bar{q} = P(d_a - d_b, R_N) - 1$.
Set $d_a := A[0]$ and $d_b := B[0]$.
if $d_a < d_b$ then return.
if $N = 0$ then Call IP_REM_NO_EXT on inputs $A$ and $B$ and return.
Set $D_q := d_a - d_b$ and $D_r := d_b - 1$.
Let $M = E[0 \ldots \bar{S}_N - 1]$ and $E' = E[\bar{S}_N \ldots e_N]$ $\{M$ points to $\bar{m}_N$ in $E[0 \ldots e_N]\}$.
Let $T_1 = W[0 \ldots t-1]$ and $T_2 = W[t \ldots 2t-1]$ and $W' = W[2t \ldots w_N]$ where $t = P(2d_N - 2, R_{N-1})$ and $d_N = M[0] = \deg_{z_N}(\bar{m}_N)$.
Set $s_c := 1 + d_a S_N$
for $k = d_a$ to $0$ by $-1$ do
   Copy $A[sc \ldots sc + S_N - 1]$ into $T_1[0 \ldots S_N - 1]$.
   Set $i := \max(0, k - D_q)$, $s_b := 1 + iS_N$ and $s_a := 1 + (k - i + d_b)S_N$.
   while $i \leq \min(D_r, k)$ do
      Call IP_MUL($N - 1, A[s_a \ldots \bar{a}], B[s_b \ldots \bar{b}], T_2, E', W'$).
      Call IP_SUB($N - 1, T_1, T_2$) $\{T_1 := T_1 - T_2\}$.
      Set $s_b := s_b + S_N$ and $s_a := s_a - S_N$.
   end while
   Call IP_REM($N - 1, T_1, M, E', W'$) {Reduce $T_1$ modulo $M = \bar{m}_N$}.
   Copy $T_1[0 \ldots S_N - 1]$ into $A[s_c \ldots s_c + S_N - 1]$.
   Set $s_c := s_c - S_N$.
end for
Set $i := D_r$ and $s_c := 1 + D_r S_N$.
while $i \geq 0$ and $A[s_c] = -1$ do Set $i := i - 1$ and $s_c := s_c - S_N$.
Set $A[0] := i$.
```

Let arrays A and B represent polynomials a and b respectively. Let $d_a = \deg_x(a)$ and $d_b = \deg_x(b)$. Array A has enough space to store $d_a + 1$ *coefficients* in R_N plus one unit of storage to store d_a. Hence the total storage is $(d_a + 1)S_N + 1$. The remainder $\bar{R}$ is of degree at most $d_b - 1$ in x, i.e. $\bar{R}$ needs storage for d_b coefficients in R_N and one unit for the degree. Similarly the quotient $\bar{Q}$ is of degree $d_a - d_b$, hence needs storage for $d_a - d_b + 1$ coefficients and one unit for the degree. Thus the remainder and the quotient together need

$d_bS_N+1+(d_a-d_b+1)S_N+1 = (d_a+1)S_N+2$. This means we are one unit of storage short if we want to store both $\bar{R}$ and $\bar{Q}$ in A. This is because this time we are storing two degrees for $\bar{Q}$ and $\bar{R}$. Our solution is that we will not store the degree of $\bar{Q}$. Any algorithm that calls IP_REM and needs both the quotient and the remainder must use $\deg_x(a) - \deg_x(b)$ for the degree of $\bar{Q}$.

After applying this algorithm the remainder $\bar{R}$ will be stored in $A[0 \dots d_bS_N]$ and the quotient $\bar{Q}$ minus the degree will be stored in $A[d_bS_N \dots (d_a+1)S_N]$. Similar to IP_MUL, the remainder operation in line 16 has been moved to outside of the main loop to let the values accumulate in T_1.

Computing (In-place) the inverse of an element in R_N

In this algorithm we assume the following in-place function:

- IP_SCAL_MUL(N, A, C, E, W): This is used for multiplying a polynomial $a \in R_N[x]$ (represented by array A) by a scalar $c \in R_N$ (represented by array C). The algorithm will multiply every coefficient of a in x by c and reduce the result modulo the minimal polynomials. It can easily be implemented using IP_MUL and IP_REM.

The algorithm computes the inverse of an element a in R_N. If the element is not invertible, then the Euclidean algorithm will compute a proper divisor of some minimal polynomial $m_i(z_i)$, a zero-divisor in R_i. The algorithm will store that zero-divisor in the space provided for the inverse and return the index i of the minimal polynomial which is reducible and has caused the zero-divisor.

Algorithm 3.6: Algorithm IP_INV: In-place inverse of an element in R_N

Require:

- N the number of field extensions.
- Array $A[0 \dots \bar{a}]$ representing the univariate polynomial $a \in R_N$. Note that $N \geq 1$ and $\bar{a} = S_N - 1$.
- Array $I[0 \dots \bar{i}]$: Space needed for storing the inverse $a^{-1} \in R_N$. Note that $\bar{i} = S_N - 1$.
- $E[0 \dots e_N]$: representing the set of minimal polynomials. Note that $e_N = S_N + 2\sum_{i=1}^{N-1} S_i$.

- $W[0 \ldots w_N]$: *the working storage* for the intermediate operations.

Ensure: The inverse of a (or a zero-divisor, if there exists one) will be computed and stored in I. If there is a zero-divisor, the algorithm will return the index k where $\bar{m}_k$ is the reducible minimal polynomial, otherwise it will return 0.

```
Let $M = E[0 \ldots \bar{S}_N - 1]$ and $E' = E[\bar{S}_N \ldots e_N]$ $\{M = \bar{m}_N\}$.
**if** $N = 1$ **then** Call IP_INV_NO_EXT on inputs $A$, $I, E, M$ and $W$ and **return.**
**if** $A[i] = 0$, for all $0 \leq i < N$ and $A[N] = 1$ $\{$ Test if $a = 1\}$ **then**
   Copy $A$ into $I$ and **return 0.**
**end if**
Let $r_1 = W[0 \ldots t-1], r_2 = W[t \ldots 2t-1], s_1 = I, s_2 = W[2t \ldots 3t-1], T = W[3t \ldots 4t-1]$, $T' = W[4t \ldots 4t + t' - 1]$ and $W' = W[4t + t' \ldots w_N]$ where $t = P(d_N, R_{N-1}) - 1 = \bar{S}_N - 1$, $t' = P(2d_N - 2, R_{N-1})$ and $d_N = M[0] = \deg_{z_N}(\bar{m}_N)$.
Copy $A$ and $M$ into $r_1$ and $r_2$ respectively.
Set $s_2[0] := -1$ $\{s_2$ represents $0\}$.
Let $Z \in \mathbb{Z}$ := IP_INV$(N - 1, A[D_a S_{N-1} + 1 \ldots \bar{a}], T, E', W')$ where $D_a = A[0] = \deg_{z_N}(a)$. $\{A[D_a S_{N-1} + 1 \ldots \bar{a}]$ represents $l = \mathrm{lc}_{z_N}(a)$ and $T$ represents $l^{-1}$.$\}$
**if** $Z > 0$ **then** Copy $T$ into $I$ and **return** $Z$.
Copy $T$ into $s_1$.
Call IP_SCAL_MUL$(N, r_1, T, E', W')$ $\{r_1$ is made monic$\}$.
**while** $r_2[0] \neq -1$ **do**
   Set $Z$ = IP_INV$(N - 1, r_2[D_{r_2} S_{N-1} + 1 \ldots \bar{a}], T, E', W')$ where $D_{r_2} = \deg_{z_N}(r_2)$.
   **if** $Z > 0$ **then** Copy $T$ into $I$ and **return** $Z$.
   Call IP_SCAL_MUL$(N, r_2, T, E', W')$ $\{r_2$ is made monic$\}$.
   Call IP_SCAL_MUL$(N, s_2, T, E', W')$.
   Set $D_q := \max(-1, r_1[0] - r_2[0])$.
   Call IP_REM$(N, r_1, r_2, E', W')$.
   Swap the arrays $r_1$ and $r_2$. $\{$Interchange only the pointers$\}$.
   Set $t_1 := r_2[r_1[0] S_{N-1}]$ and set $r_2[r_1[0] S_{N-1}] := D_q$.
   Call IP_MUL$(N - 1, q, s_2, T', E', W')$ where $q = r_2[r_1[0] S_{N-1} \ldots \bar{a}]$.
   Call IP_REM$(N - 1, T', M, E', W')$ and then IP_SUB$(N - 1, s_1, T')$. $\{s_1 := s_1 - q s_2\}$
   Set $r_2[r_1[0] S_{N-1}] := t_1$.
   Swap the arrays $s_1$ and $s_2$. $\{$Interchange only the pointers$\}$.
**end while**
```

```
27: **if** $r_1[i] = 0$ for all $0 \leq i < N$ and $r_1[N] = 1$ **then**
28:     Copy $s_1$ into $I$ $\{r_1 = 1$ and $s_1$ is the inverse$\}$ and **return** 0.
29: **else**
30:     Copy $r_1$ into $I$ $\{r_1 \neq 1$ is the zero-divisor$\}$ and **return** $N - 1$ $\{\bar{m}_{N-1}$ is reducible$\}$.
31: **end if**
```

As discussed in Section 3.1.2, IP_REM will not store the degree of the quotient of a divided by b hence in line 21 we explicitly compute and set the degree of the quotient before using it to compute $s_1 := s_1 - qs_2$ in lines 22 and 23. Here $r_2[r_1[0]S_{N-1} \ldots \bar{a}]$ is the quotient of dividing r_1 by r_2 in line 19.

In-place GCD Computation

In the following algorithm we compute the GCD of $a, b \in R_N[x]$ using the monic Euclidean algorithm. This is the main subroutine used to compute univariate images of a GCD in $L[x]$ for the algorithm in [70] and images of a multivariate GCD over an algebraic function field for our algorithm in [31]. Note, since $m_i(z_i)$ may be reducible modulo p, R_N is is not necessarily a field, and therefore, the monic Euclidean algorithm may encounter a zero-divisor in R_N when calling subroutine IP_INV.

ALGORITHM 3.7: Algorithm IP_GCD: In-place GCD Computation

```
**Require:**
  • $N$ the number of field extensions.
  • Arrays $A[0 \ldots \bar{a}]$ and $B[0 \ldots \bar{b}]$ representing univariate polynomials $a, b \neq 0 \in R_N[x]$ ($R_N = \mathbb{Z}_p[z_1, \ldots, z_N] / \langle \bar{m}_1, \ldots, \bar{m}_N \rangle$) where $d_a = \deg_x(a) \geq d_a = \deg_x(b)$ and $A, B \neq 0$. Note that $b$ is monic and $\bar{a} = P(d_a, R_N) - 1$ and $\bar{b} = P(d_b, R_N) - 1$.
  • $E[0 \ldots e_N]$ : representing the set of minimal polynomials where $e_N = S_N + 2\sum_{i=1}^{N-1} S_i$.
  • $W[0 \ldots w_N]$ : *the working storage* for the intermediate operations.
**Ensure:** If there exist a zero-divisor, it will be stored in $A$ and the index of the reducible minimal polynomial will be returned. Otherwise the monic GCD $g = \gcd(a, b)$ will be stored in $A$ and 0 will be returned.
1: **if** $N = 0$ **then** CALL IP_GCD_NO_EXT on inputs $A$ and $B$ and **return 0.**
```

```
2: Set $d_a := A[0]$ and $d_b := B[0]$.
3: Let $r_1$ and $r_2$ point to $A$ and $B$ respectively.
4: Let $I = W[0 \dots t-1]$ and $W' = W[t \dots w_N]$ where $t = \bar{S}_N - 1 = S_N + S_{N-1} - 1$.
5: Let $Z$ be the output of IP_INV($N, r_1[1 + r_1[0]S_N \dots \bar{a}], I, E, W'$).
6: if $Z > 0$ then Copy $I$ into $A$ and return $Z$.
7: Call IP_SCAL_MUL($N, r_1, I, E, W'$).
8: while $r_2[0] \neq -1$ do
9:    Let $Z$ be the output of IP_INV($N, r_2[1 + r_2[0]S_N \dots \bar{b}]$, $I$, $E, W'$).
10:   if $Z > 0$ then Copy $I$ into $A$ and return $Z$.
11:   Call IP_SCAL_MUL($N, r_2, I, E, W'$).
12:   Call IP_REM($N, r_1, r_2, E, W'$).
13:   Swap $r_1$ and $r_2$ {interchange pointers}.
14: end while
15: Copy $r_1$ into $A$.
16: return 0.
```

Similar to the algorithm IP_INV, if there exists a zero-divisor, i.e. the leading coefficient of one of the polynomials in the polynomial remainder sequence is not invertible, in steps 6 and 10 the algorithm stores the zero-divisor in the space provided for a and returns Z the index of the minimal polynomial which is reducible and has caused the zero-divisor.

3.1.3 Working Space

In this section we will determine recurrences for the exact amount of working storage w_N needed for each operation introduced in the previous section. Recall that $d_i = \deg_{z_i}(\bar{m}_i)$ is the degree of the ith minimal polynomial which we may assume is at least 2. Also S_i is the space needed to store an element in R_i and we have $S_{i+1} = d_{i+1}S_i + 1$ and $S_1 = d_1 + 1$.

Lemma 3.8. $S_N > 2S_{N-1}$ for $N > 1$.

Proof. We have $S_N = d_N S_{N-1} + 1$ where $d_N = \deg_{z_N}(\bar{m}_N)$. Since $d_N \geq 2$ we have $S_N \geq 2S_{N-1} + 1 \Rightarrow S_N > 2S_{N-1}$. □

Lemma 3.9. $\sum_{i=1}^{N-1} S_i < S_N$ for $N > 1$.

Proof. (by induction on N). For $N = 2$ we have $\sum_{i=1}^{1} S_i = S_1 < S_2$. For $N = k+1 \geq 2$ we have $\sum_{i=1}^{k} S_i = S_k + \sum_{i=1}^{k-1} S_i$. By induction we have $\sum_{i=1}^{k-1} S_i < S_k$ hence $\sum_{i=1}^{k} S_i <$

$S_k + S_k = 2S_k$. Using Lemma 3.8 we have $2S_k < S_{k+1}$ hence $\sum_{i=1}^{k} S_i < 2S_k < S_{k+1}$ and the proof is complete. □

Corollary 3.10. $\sum_{i=1}^{N} S_i < 2S_N$ for $N > 1$.

Lemma 3.11. $P(2d_N - 2, R_{N-1}) = 2S_N - S_{N-1} - 1$ for $N > 1$.

Proof. We have $P(2d_N - 2, R_{N-1}) = (2d_N - 1)S_{N-1} + 1 = 2d_N S_{N-1} - S_{N-1} + 1 = 2(d_N S_{N-1} + 1) - S_{N-1} - 1 = 2S_N - S_{N-1} - 1$. □

Multiplication and Division Algorithms

Let $M(N)$ be the amount of working storage needed to multiply $a, b \in R_N[x]$ using the algorithm IP_MUL. Similarly let $Q(N)$ be the amount of working storage needed to divide a by b using the algorithm IP_REM. The working storage used in lines 5,11 and 15 of algorithm IP_MUL and lines 6,12 and 16 of algorithm IP_REM is

$$M(N) = 2P(2d_N - 2, R_{N-1}) + \max(M(N-1), Q(N-1)) \text{ and } \tag{3.1}$$

$$Q(N) = 2P(2d_N - 2, R_{N-1}) + \max(M(N-1), Q(N-1)). \tag{3.2}$$

Comparing equations (3.1) and (3.2) we see that $M(N) = Q(N)$ for any $N \geq 1$. Hence

$$M(N) = 2P(2d_N - 2, R_{N-1}) + M(N-1). \tag{3.3}$$

Simplifying (3.3) gives $M(N) = 2S_N - 2N + 2\sum_{i=1}^{N} S_i$. Using Corollary 3.10 we have

Theorem 3.12. $M(N) = Q(N) = 2S_N - 2N + 2\sum_{i=1}^{N} S_i < 6S_N$.

Remark 3.13. When calling the algorithm IP_MUL to compute $c = a \times b$ where $a, b \in R[x]$, we should use a working storage array $W[0 \ldots w_n]$ such that $w_n \geq M(N)$. Since $M(N) < 6S_N$, the working storage must be big enough to store only six coefficients in L_p.

Let $C(N)$ denote the working storage needed for the operation IP_SCAL_MUL. It is easy to show that $C(N) = M(N-1) + P(2d_N - 2, R_{N-1}) < M(N)$.

Inversion

Let $I(N)$ denote the amount of working storage needed to invert $c \in R_N$. In lines 6, 9, 12, 14, 16, 17, 19 , 22 and 23 of algorithm IP_INV we use the working storage. We have

$$I(N) = 4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) + \max(I(N-1), M(N-1), Q(N-1)). \quad (3.4)$$

But we have $M(N-1) = Q(N-1)$, hence

$$I(N) = 4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) + \max(I(N-1), M(N-1)). \quad (3.5)$$

Lemma 3.14. For $N \geq 1$, we have $M(N) < I(N)$.

Proof. The proof is by contradiction. Assume $M(N) \geq I(N)$. Using (3.5) we have $I(N) = 4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) + M(N-1)$. On the other hand using (3.3) we have $M(N) = 2P(2d_N - 2, R_{N-1}) + M(N-1)$. We assumed $I(N) \leq M(N)$ hence we have $4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) + M(N-1) \leq 2P(2d_N - 2, R_{N-1}) + M(N-1)$ thus $4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) \leq 2P(2d_N - 2, R_{N-1}) \Rightarrow 6S_N + 3S_{N-1} - 1 \leq 4S_N - 2S_{N-1} - 2$ which is a contradiction. Thus $I(N) > M(N)$. □

Using Equation (3.4) and Lemma 3.14 we conclude that $I(N) = 4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) + I(N-1)$. Simplifying this yields:

Theorem 3.15. $I(N) = 4\sum_{i=1}^{N} P(d_i, R_{i-1}) + \sum_{i=1}^{N} P(2d_i - 2, R_{i-1}) = 4\sum_{i=1}^{N}(S_i + S_{i-1}) + \sum_{i=1}^{N}(2S_i - S_{i-1} - 1) = 6S_N + 9\sum_{i=1}^{N-1} S_i - N$.

Using Lemma 3.8 an upper bound for $I(N)$ is $I(N) < 6S_N + 9S_N = 15S_N$.

GCD Computation

Let $G(N)$ denote the working storage needed to compute the GCD of $a, b \in R_N[x]$. In lines 4,5,7,9,11 and 12 of algorithm IP_GCD we use the working storage. We have $G(N) = \bar{S}_N + \max(I(N), C(N), Q(N))$. Lemma 3.14 states that $I(N) > M(N) = Q(N) > C(N)$ hence

$$G(N) = \bar{S}_N + I(N) = S_N + S_{N-1} + 6S_N + 9\sum_{i=1}^{N-1} S_i - N = 7S_N + S_{N-1} + 9\sum_{i=1}^{N-1} S_i - N.$$

Since $I(N) < 15S_N$, we have an upper bound on $G(N)$:

Theorem 3.16. $G(N) = S_N + S_{N-1} + I(N) < S_N + S_{N-1} + 15S_N < 17S_N$.

Remark 3.17. The constants 6, 15 and 17 appearing in Theorems 6, 9 and 10 respectively, are not the best possible. One can reduce the constant 6 for algorithm IP_MUL if one also uses the space in the output array C for working storage. We did not do this because it complicates the description of the algorithm and yields no significant performance gain.

3.1.4 Benchmarks

We have compared our C library with the Magma (see [4]) computer algebra system. The results are reported in Table 3.1. For our benchmarks we used $p = 3037000453$, two field extensions with minimal polynomials $\bar{m}_1$ and $\bar{m}_2$ of varying degrees d_1 and d_2 but with $d = d_1 \times d_2 = 60$ constant so that we may compare the overhead for varying d_1. We choose three polynomials a, b, g of the same degree d_x in x with coefficients chosen from R at random. The data in the fifth and sixth columns (labelled IP_MUL and MAG_MUL) are the times (in CPU seconds) for computing both $f_1 = a \times g$ and $f_2 = b \times g$ using IP_MUL and Magma version 2.15 respectively. Similarly, the data in the seventh and eighth columns (labelled IP_REM and MAG_REM) are the times for computing both quo(f_1, g) and quo(f_2, g) using IP_REM and Magma respectively. Finally the data in the ninth and tenth columns (labelled IP_GCD and MAG_GCD) are the times for computing gcd(f_1, f_2) using IP_GCD and Magma respectively. The data in the column labeled $\#f_i$ is the number of terms in f_1 and f_2.

The timings in Table 3.1 for *in-place* routines show that as the degree d_x doubles from 40 to 80, the time consistently goes up by a factor of 4 indicating that the underlying algorithms are all quadratic in d_x. This is not the case for Magma because Magma is using a sub-quadratic algorithm for multiplication. We describe the algorithm used by Magma ([65]) briefly. To multiply two polynomials $a, b \in L_p[x]$ Magma first multiplies a and b as polynomials in $\mathbb{Z}[x, z_1, \ldots, z_r]$. It then reduces their product modulo the ideal $\langle m_1, \ldots, m_r, p\rangle$. To multiply in $\mathbb{Z}[x, z_1, \ldots, z_r]$, Magma evaluates each variable successively, beginning with z_r then ending with x, at integers $k_r, \ldots, k_1, k_0$ which are powers of the base of the integer representation which are sufficiently large so that that the product of the two polynomials $a(x, z_1, \ldots, z_r) \times b(x, z_1, \ldots, z_r)$ can be recovered from the product of the two (very) large integers $a(k_0, k_1, \ldots, k_r) \times b(k_0, k_1, \ldots, k_r)$. The reason to evaluate at a power of the integer base is so that evaluation and recovery can be done in linear time.

Table 3.1: Timings in CPU seconds on an AMD Opteron 254 CPU running at 2.8 GHz

d_1	d_2	d_x	$\#f_i$	IP_MUL	MAG_MUL	IP_REM	MAG_REM	IP_GCD	MAG_GCD
2	30	40	2460	0.124	0.050	0.123	0.09	0.384	2.26
3	20	40	2460	0.108	0.054	0.106	0.11	0.340	2.35
4	15	40	2460	0.106	0.056	0.106	0.10	0.327	2.39
6	10	40	2460	0.106	0.121	0.105	0.14	0.328	5.44
10	6	40	2460	0.100	0.093	0.100	0.37	0.303	7.84
15	4	40	2460	0.097	0.055	0.095	0.17	0.283	3.27
20	3	40	2460	0.092	0.046	0.091	0.14	0.267	2.54
30	2	40	2460	0.087	0.038	0.087	0.10	0.242	1.85
2	30	80	4860	0.477	0.115	0.478	0.27	1.449	9.41
3	20	80	4860	0.407	0.127	0.409	0.27	1.304	9.68
4	15	80	4860	0.404	0.132	0.406	0.28	1.253	9.98
6	10	80	4860	0.398	0.253	0.400	0.35	1.234	22.01
10	6	80	4860	0.380	0.197	0.381	0.86	1.151	31.57
15	4	80	4860	0.365	0.127	0.364	0.40	1.081	13.49
20	3	80	4860	0.353	0.109	0.353	0.33	1.030	10.59
30	2	80	4860	0.336	0.086	0.337	0.26	0.932	7.83

In this way polynomial multiplication in $\mathbb{Z}[x, z_r, \ldots, z_1]$ is reduced to a single (very) large integer multiplication which is done using the FFT. This, note, may not be efficient if the polynomials $a(x, z_1, \ldots, z_r)$ and $b(x, z_1, \ldots, z_r)$ are sparse.

Table 3.1 shows that our in-place GCD algorithm is a factor of 6 to 27 times faster than Magma's GCD algorithm. Since both algorithms use the Euclidean algorithm, this shows that our in-place algorithms for arithmetic in L_p are efficient. This is the gain we sought to achieve. The reader can observe that as d_1 increases, the timings for IP_MUL decrease which shows there is still some overhead for α_1 of low degree.

Optimizations in the implementation

In modular algorithms, multiplication in $\mathbb{Z}_p$ needs to be coded carefully. This is because hardware integer division (`%p` in C) is much slower than hardware integer multiplication. One can use Peter Montgomery's trick (see [59]) to replace all divisions by p by several cheaper operations for an overall gain of typically a factor of 2. Instead, we use the following scheme which replaces most divisions by p in the multiplication subroutine for $\mathbb{Z}_p[x]$ by at most one subtraction. We use a similar scheme for the division in $\mathbb{Z}_p[x]$. This makes GCD computation in $L_p[x]$ more efficient as well. We observed a gain of a factor of 5 on average for the GCD computations in our benchmarks.

The following C code explains the idea. Suppose we have two polynomials $a, b \in \mathbb{Z}_p[x]$ where $a = \sum_{i=0}^{d_a} a_i x^i$ and $b = \sum_{j=0}^{d_b} b_j x^j$ where $a_i, b_j \in \mathbb{Z}_p$. Suppose the coefficients a_i and b_i are stored in two Arrays A and B indexed from 0 to d_a and 0 to d_b respectively. We assume elements of $\mathbb{Z}_p$ are stored as signed integers and an integer x in the range $-p^2 < x < p^2$ fits in a machine word. The following computes $c = a \times b = \sum_{k=0}^{d_a+d_b} c_k x^k$.

```
M = p*p;
d_c = d_a+d_b;
for( k=0; k<=d_c; k++ ) {
   t = 0;
   for( i=max(0,k-d_b); i <= min(k,d_a); i++ )
   {
      if( t<0 ); else t = t-M;
      t = t+A[i]*B[k-i];
   }
   t = t % p;
   if( t<0 ) t = t+p;
   C[k] = t;
}
```

The trick here is to put t in the range $-p^2 < t \leq 0$ by subtracting p^2 from it when it is positive so that we can add the product of two integers $0 \leq a_i, b_{k-i} < p$ to t without overflow. Thus the number of divisions by p is linear in d_c, the degree of the product. One can further reduce the number of divisions by p. In our implementation, when multiplying elements $a, b \in \mathbb{Z}_p[z][x]/\langle m(z) \rangle$ we multiply $a, b \in \mathbb{Z}_p[z][x]$ without division by p before dividing by $m(z)$.

Note that the statement `if( t<0 ); else t = t-M;` is done this way rather than the more obvious `if( t>0 ) t = t-M;` because it is faster. The reason is that $t < 0$ holds about 75% of the time and the code generated by the newer compilers is optimized for the case the condition of an if statement is true. If one codes the if statement using `if( t>0 ) t = t-M;` instead, we observe a loss of a factor of 2.6 on an Intel Core i7, 2.3 on an Intel Core 2 duo, and 2.2 on an AMD Opteron for the above code.

3.1.5 Remarks

Our C library of in-place routines has been integrated into Maple 14 for use in the GCD algorithms in [71] and [31]. These algorithms compute GCDs of polynomials in $K[x_1, x_2, \ldots, x_n]$ over an algebraic function field K in parameters $t_1, t_2, \ldots, t_k$ by evaluating the parameters and variables except x_1 and using rational function interpolation to recover the GCD. This results in many GCD computations in $L_p[x_1]$. In many applications, K has field extensions of low degree, often quadratic or cubic. Our C library is available on our website at

`http://www.cecm.sfu.ca/CAG/code/ASCM09/inplace.c`

The code used to generate the Magma timings in Section 3.1.4 is available in the file

`http://www.cecm.sfu.ca/CAG/code/ASCM09/magma.txt`

The algebraic number field $\mathbb{Q}(\alpha_1, \ldots, \alpha_r)$ can be transformed to a field with only one extension: $\mathbb{Q}(\alpha)$. The algebraic extension α can be a linear combination of $\alpha_1, \ldots, \alpha_r$ and the minimal polynomial of α can be found by computing a series of resultants. An extension point to our work here is to explore the efficiency of working with an algebraic number field of only one extension.

In [53], Xin, Moreno Maza and Schost develop asymptotically fast algorithms for multiplication in L_p based on the FFT and use their algorithms to implement the Euclidean algorithm in $L_p[x]$ for comparison with Magma and Maple. The authors obtain a speedup for L of sufficiently large degree d. Our results in this chapter are complementary in that we sought to improve arithmetic when L has relatively low degree.

3.2 Eliminating the Trial Divisions

In Section 1.2.5 we introduced the problem of rational function reconstruction. Let $m, u \in \mathbb{Z}_p[x]$ where $0 \le \deg(u) < \deg(m)$. The problem of rational function reconstruction is given m and u, find $n, d \in \mathbb{Z}_p[x]$ such that

$$\frac{n}{d} \equiv u \bmod m,$$

satisfying $\mathrm{lc}_x(d) = 1$ and $\gcd(m, d) = \gcd(n, d) = 1$. To interpolate $\frac{n}{d}$, we can use the Extended Euclidean algorithm to find all the solutions satisfying $\deg(n) + \deg(d) < \deg(m)$ up to multiplication by scalars.

The MQRR algorithm is designed so that when it succeeds, the output of the SparseModGcd algorithm is the correct image of the GCD with high probability. This will be accomplished by having one more evaluation point than needed to interpolate a parameter or a variable. Also in Zippel's sparse interpolation, a wrong form for the GCD will be detected with high probability by using one more evaluation than necessary. Hence the trial divisions (both in positive and zero characteristic) will rarely fail in practice.

This motivates us to prove that if we omit the trial divisions in positive characteristic in Step 12 of Algorithm 3.2 (computing $\gcd(f_1(\alpha), f_2(\alpha)) \bmod p$, where p is the prime, f_1 and f_2 are the input polynomials and α is the evaluation point), the SparseModGcd algorithm will *terminate* and output the correct GCD $g = \gcd(f_1, f_2)$. To be able to prove this, we must show that the probability that the MQRR algorithm makes an error is low. Recall from Section 1.2.5 that the MQRR algorithm will output $\frac{r_i}{t_i}$ for q_{i+1} the quotient of maximal degree provided $\deg q_{i+1} \geq T$ for some $T > 1$. Here we will prove, the probability that MQRR fails is at most d_m^T/p^{T-1} where $d_m = \deg(m)$. In our implementation of MQRR, we choose $T = 2$. For this case the probability of failure is at most d_m^2/p. For a given ϵ, if we choose a prime $p > d_m^2 \times (\frac{1}{\epsilon})$ then the probability of failure is at most ϵ.

Recall that on inputs m and u, the extended Euclidean algorithm computes a sequence of triples s_i, t_i, r_i satisfying $s_i m + t_i u = r_i$. Let $n, d \in \mathbb{Z}_p[x]$. Without loss of generality, assume that $\mathrm{lc}_x(d) = 1$. For evaluation points $\alpha_1 \in \mathbb{Z}_p, \ldots, \alpha_k \in \mathbb{Z}_p$, let $m_k = \prod_{i=1}^k (x - \alpha_i)$. We have $\deg(m_k) = k$. Let d_k^{-1} be the inverse of d modulo $\langle m_k, p\rangle$. Define $u_k = (n d_k^{-1}) \bmod m_k$. We have $u_k \equiv \frac{n}{d} \bmod \langle m_k, p\rangle$. Suppose we apply the extended Euclidean algorithm on inputs u_k and m_k modulo p. Let the i'th remainder be r_i and the i'th quotient be q_i. Let $S_k = \{(\bar{r}_i, \bar{t}_i) : 1 \leq i \leq k\}$ where $\bar{r}_i = r_i/\mathrm{lc}(t_i) \bmod p$ and $\bar{t}_i = t_i/\mathrm{lc}(t_i) \bmod p$.

Lemma 3.18. (See [57, 79, 80]) If $\deg(r_i) + \deg(t_i) < \deg(m_k) = k$ and $r_i/t_i \equiv n/d \bmod m_k$ and $\mathrm{lc}(t_i) = 1$ then $(r_i, t_i) \in S_k$.

Lemma 3.19. (See [57]) Let $k \in \mathbb{N}$ such that $k > (\deg(n) + \deg(d))$. Suppose we apply the extended Euclidean algorithm to u_k and m_k and we get $r_i/t_i \equiv n/d \bmod m_k$ for $1 \leq i \leq k$. We have $\deg(q_{i+1}) = k - \deg(r_i) - \deg(t_i)$.

Lemma 3.20. Suppose the MQRR algorithm on inputs u_k, m_k and $T > 1$ succeeds and returns $(r_i, t_i) \in S_k$ ($\mathrm{lc}(t_i) = 1$). For all j where $k - T < j$ we have $(r_i, t_i) \in S_j$.

Proof. The MQRR algorithm succeeds hence $r_i/t_i \equiv n/d \bmod m_k$ and $deg(q_{i+1}) \geq T$, so using Lemma 3.19, $k = \deg(m_k) \geq T + \deg(r_i) + \deg(t_i) \Rightarrow k - T + 1 > \deg(r_i) + \deg(t_i)$.

But $j > k-T$, hence $j \geq k-T+1 > \deg(r_i)+\deg(t_i)$. Also $r_i/t_i \equiv n/d \bmod m_k \Rightarrow r_i/t_i \equiv n/d \bmod m_j$, thus using Lemma 3.18, $(r_i, t_i) \in S_j$. □

Theorem 3.21. *Let* $\deg(n) + \deg(d) < k$. *Suppose the MQRR algorithm on inputs* u_k, m_k *(satisfying* $\frac{n}{d} \equiv u_k \bmod m_k$*) and* $T > 1$ *succeeds and returns* n'/d' $(lc(d') = 1)$. *We have*

$$Prob(\frac{n'}{d'} \neq \frac{n}{d}) \leq \frac{k^T}{p^{T-1}}.$$

Proof. Suppose $n'/d' \neq n/d$. There exists $1 \leq i \leq k$ such that $(r_i, t_i) \in S_k$ and $r_i = n'$ and $t_i = d'$. Using Lemma 3.19 we have $\deg(q_{i+1}) = k - \deg(r_i) - \deg(t_i)$ but $\deg(q_{i+1}) \geq T$ hence

$$k - \deg(r_i) - \deg(t_i) > T \Rightarrow k - T > \deg(r_i) + \deg(t_i). \tag{3.6}$$

Lemma 3.20 indicates that $(r_i, t_i) \in S_j$ for $k-T+1 \leq j \leq k$. Let $z = dr_i - nt_i \neq 0$. We have $n'/d' \equiv n/d \bmod m_k$ hence $z \equiv 0 \bmod m_k$. That is $x - \alpha_j \mid z$ for $j \leq k$. Let $l = k - T + 1$ and $z' = \frac{z}{m_l}$. We have

$$\begin{aligned}
&\deg(z') = \deg(dr_i - nt_i) - \deg(m_l) = \\
&\max(\deg(d) + \deg(r_i), \deg(n) + \deg(t_i) - l = \\
&\max(\deg(d) + \deg(r_i), \deg(n) + \deg(t_i) - (k - T + 1) < \\
&\max(\deg(d) + \deg(r_i), \deg(n) + \deg(t_i) - (\deg(r_i) + \deg(t_i)) = \\
&\max(\deg(d) - \deg(t_i), \deg(n) - \deg(r_i)) < k.
\end{aligned}$$

On the other hand, for random evaluations α_j $(l < j \leq k)$ we have $x - \alpha_j \mid z'$. That is α_j is a root of z'. Since $\deg(z') < k$, this happens with probability at most $\frac{k^{k-l}}{p^{k-l}} = \frac{k^{T-1}}{p^{T-1}}$. Since $1 \leq i \leq k$, the probability that $n'/d' \neq n/d$ is at most

$$k \times \frac{k^{T-1}}{p^{T-1}} = \frac{k^T}{p^{T-1}}.$$

The proof is complete. □

Now suppose that MQRR succeeds and outputs g_i. In SparseModGcd, we will choose the form of the GCD g_f based on $\tilde{g}_i$ which is the primitive part of $\text{den}(g_i)g_i$. We will use g_f to compute other images of the GCD using Zippel's sparse interpolation algorithm. Let $g_f = A_1M_1 + A_2M_2 + \cdots + A_tM_t$ where A_i's are unknown integer coefficients and M_i is a monomial. Let N the number of equations we need to solve for all the unknowns. We choose $N+1$ evaluations, to obtain one more equation than necessary. Now suppose g_f is wrong, i.e. the output of MQRR is wrong. We prove that the probability that Zippel's sparse interpolation succeeds with a wrong form is at most $\frac{nd}{p}$ where d bounds the degree of the GCD in all the parameters and variables and n is the number of variables and parameters.

Theorem 3.22. *Let $g = \gcd(f_1, f_2) \in \mathbb{F}[x_1, \ldots, x_n]$ and $\deg_{x_i}(g) \leq d$. Provided we use one more evaluation point than necessary in Zippel's sparse interpolation, the probability of not detecting a wrong form and returning a wrong result is at most $\frac{nd}{p}$.*

Proof. Let g_i be the correct image of the GCD that we wish to interpolate. Let the set of evaluation points be $\{\alpha_1, \ldots, \alpha_{N+1}\} \subset \mathbb{Z}_p^{N+1}$. Suppose we need N equations to solve for all the unknowns in g_f. Assume that using the first N equations we solve for all the unknowns. We substitute the values for these unknowns in g_f and obtain $\bar{g}_i$. Suppose this polynomial is not the true image of the GCD. i.e. $g_i \neq \bar{g}_i$ but $g_i(\alpha_j) = \bar{g}_i(\alpha_j)$ for $1 \leq j \leq N$. Let $\bar{z} = g_i - \bar{g} \neq 0$. For the last evaluation point α_{N+1}, we get a new equation. If the values for the unknowns do not satisfy this new equation, then we know that the form is wrong. So assume that the values satisfy the new equation, i.e. $g_i(\alpha_{N+1}) = \bar{g}_i(\alpha_{N+1})$ which means that α_{N+1} must be a root of $\bar{z} \neq 0$. We have $\deg(\bar{z}) \leq nd$ hence using Schwartz Lemma 1.7, the probability that $\alpha_{N+1} \in \mathbb{Z}_p$ is a root of $\bar{z}$ is at most $\frac{nd}{p}$. □

Theorems 3.21 and 3.22 prove that if we eliminate the trial divisions in positive characteristic, we can make the probability of error in the MQRR algorithm and Zippel's sparse interpolation arbitrarily small by choosing bigger primes. The SparseModGcd algorithm does one trial division at the end of the algorithm to prove the correctness of the result. If this fails the algorithm will restart by choosing a different prime and different evaluation points. Also Theorems 3.21 and 3.22 prove that eventually the MQRR algorithm will succeed with the correct image of the GCD and the sparse interpolation will succeed, hence the algorithm will eventually terminate with the correct GCD computed.

3.3 The Normalization Problem

As discussed in Chapter 1, in [9] it is mentioned that if the evaluation point and the prime do not introduce any *unlucky content*, then the number of univariate images needed is

$$U = \max(\frac{N}{T-1}, n_{max}), \tag{3.7}$$

where $N = (\sum_{i=1}^{T} n_i) - 1$ is the number of unknowns in the assumed form of the GCD, n_i is the number of terms in the i'th coefficient and n_{max} is the maximum number of terms in any coefficient of the GCD in the main variable x, i.e. $n_{max} = \max(n_1, n_2, \ldots, n_T)$.

Here we will show that having $U = \max(\frac{N}{T-1}, n_{max})$ evaluation points does not necessarily provide enough images to solve for all the unknowns even though there is no unlucky content. We will give an example.

Example 3.23. Let $f_1, f_2 \in \mathbb{Z}[x, y]$ have GCD $g = (y^2+1)x^2 - (y^3+y)x + (y^3-2y+7)$. Suppose we have computed the form of the GCD, $g_f = (Ay^2+B)x^2 + (Cy^3+Dy)x + (Ey^3+Fy+G)$ and we want to compute $g \bmod p$ where $p = 17$ is the prime. Based on the formula in Equation 3.7 the number of evaluation points needed is $\max(\frac{6}{2}, 3) = 3$. Suppose we choose the evaluation points $y = 1, y = 7$ and $y = 15$. We will get the following system of equations

$$(A+B)x^2 + (C+D)x + (E+F+G) = x^2 + 16x + 3 \quad [y=1],$$
$$(15A+B)x^2 + (3C+7D)x + (3E+7F+G) = m_2(x^2+10x+4) \quad [y=7],$$
$$(4A+B)x^2 + (9C+15D)x + (9E+15F+G) = m_3(x^2+2x+4) \quad [y=15].$$

We solve this system of equations modulo $p = 17$, but the system is under-determined. In fact no matter what evaluation points we choose the system of linear equation with three points is always under-determined. If we choose one more point, say $y = 8$, we will get a new image of the GCD and the system will be determined and we will have the values for the unknown coefficients.

Example 3.23 illustrates that the formula in Equation 3.7 does not necessarily gives us enough points to solve for the unknown coefficients. Also it is not true that if the GCD has a content, then the system of linear equations will never be determined. Here is an example.

Example 3.24. Let $g = (y^6-1)x^2 + (y^7-1)x + (y^9-1)$. The content of the GCD in the main variable x is $\text{cont}_x(g) = y - 1$. Let $p = 17$ and we want to compute $g \bmod p$. The form for the GCD is $g_f = (Ay^6 + B)x^2 + (Cy^7 + D)x + (Ey^9 + F)$. Suppose we choose the evaluation points $y = 2, y = 3$ and $y = 5$. We will get the following system of linear equations.

$$\begin{gathered}(13\,A + B)\,x^2 + (9\,C + D)\,x + 2\,E + F = x^2 + 12x + 10, \\ (15\,A + B)\,x^2 + (11\,C + D)\,x + 14\,E + F = m_2(x^2 + 4x + 7), \\ (2\,A + B)\,x^2 + (10\,C + D)\,x + 12\,E + F = m_3(x^2 + 9x + 11).\end{gathered}$$

This system of linear equations is determined and we have

$$\{A = 10, B = 7, C = 10, D = 7, E = 10, F = 7, m2 = 4, m3 = 10\}.$$

Our New Method

Let $f_1, f_2 \in F[x_1, \ldots, x_n]$ where F is a field. Let $g = \gcd(f_1, f_2) = \sum_{i=1}^{T} C_i M_i$ where M_i is a monomial in the main variable x_1 and C_i is in $x_2, \ldots, x_n$ with unknown coefficients. Assume that $C_1 M_1$ is the leading term. The univariate images of the GCD that we obtain from the Euclidean algorithm are images of $g' = \frac{g}{C_1} = M_1 + \frac{C_2}{C_1} M_2 + \cdots + \frac{C_T}{C_1} M_T$ evaluated at the evaluation point. Note that we can always normalize the univariate image of the GCD based on any coefficient C_i and not only the leading coefficient C_1.

Let n_i be the number of terms in C_i. Let $h_{ij} = \gcd(C_i, C_j), A_{ij} = C_i/h_{ij}$ and $B_{ij} = C_j/h_{ij}$. We have $\frac{C_i}{C_j} = \frac{A_{ij}}{B_{ij}}$. Let U_{ij} and V_{ij} be the number of terms in A_{ij} and B_{ij} respectively.

Lemma 3.25. We can solve for the unknowns in C_i and C_j using $n_i + n_j - 1$ evaluation points if and only if $n_i + n_j \le U_{ij} + V_{ij}$.

Proof. There are exactly $n_i + n_j$ unknown coefficients in C_i and C_j. We can always fix one of the unknown coefficients to be 1, since C_i/C_j is only unique up to a scalar. So there are $n_i + n_j - 1$ unknowns. Each univariate image of the GCD provides one image of C_i/C_j which can be interpolated with $U_{ij} + V_{ij} - 1$ points (provided we know the form for A_{ij} and B_{ij}). So the maximum number of independent equations we get using the images of C_i/C_j is $U_{ij} + V_{ij} - 1$. Hence we can solve for the unknowns if and only if we have at least $n_i + n_j - 1$ independent equations. We have this many equations if and only if $n_i + n_j - 1 \le U_{ij} + V_{ij} - 1$ or equivalently $n_i + n_j \le U_{ij} + V_{ij}$. □

Lemma 3.26. We will have enough independent equations to solve for all the unknowns in the form of the GCD if there exist $1 \leq i \neq j \leq T$ such that $n_i + n_j \leq U_{ij} + V_{ij}$

Proof. Suppose for $1 \leq i \neq j \leq T$ we have $n_i + n_j \leq U_{ij} + V_{ij}$. We can scale the images of the GCD based on the j'th coefficient to obtain univariate images of

$$g_f = \frac{C_1}{C_j} M_1 + \cdots + \frac{C_{j-1}}{C_j} M_{j-1} + M_j + \frac{C_{j+1}}{C_j} M_{j+1} + \cdots + \frac{C_T}{C_j} M_T.$$

Since $n_i + n_j \leq U_{ij} + V_{ij}$, using Lemma 3.25 we can solve for the unknown coefficients in C_i and C_j. Then we can multiply each univariate image by the image of C_j and solve for every unknown in C_k independently using the integer coefficients of the univariate images of the GCD. □

Lemma 3.26 is a special case of the following theorem.

Theorem 3.27. *We will have enough independent equations to solve for all the unknowns in the form of the GCD if and only if there exist $S = \{i_1, i_2, \ldots, i_k\} \subseteq \{1, 2, \ldots, T\}$ such that i_j's are distinct and*

$$\sum_{j=1}^{k} (n_{i_j}) - 1 \leq \sum_{j=2}^{k} (U_{i_j i_1} + V_{i_j i_1} - 1).$$

Proof. Suppose for the set $S = \{i_1, i_2, \ldots, i_k\} \subseteq \{1, 2, \ldots, T\}$, we have $\sum_{j=1}^{k} (n_{i_j}) - 1 \leq \sum_{j=2}^{k} (U_{i_j i_1} + V_{i_j i_1} - 1)$. Suppose we scale the univariate images of the GCD based on i_1'th coefficient. For each $J \in S - i_1$, the J'th coefficient in the GCD will contribute at most $(U_{J i_1} + V_{J i_1} - 1)$ independent equations. Also the equations that we get from two different coefficients in the GCD are independent because they have different unknowns. Hence the total number of equations that we get from the coefficients with indices in $S - \{i_1\}$ is $\sum_{j=2}^{k} (U_{i_j i_1} + V_{i_j i_1} - 1)$ which is greater than the total number of unknowns which is $\sum_{j=1}^{k} (n_{i_j}) - 1$ and hence we can solve for all the unknowns in C_J, where $J \in S$ and then solve the linear systems for the rest of coefficients (with indices in $\{1, 2, \ldots, T\} - S$) independently. Conversely, for $k = T$, we have $\sum_{j=2}^{T} (U_{i_j i_1} + V_{i_j i_1} - 1)$ is the *maximum* number of independent equations that we get using the univariate images, no matter how many evaluation points, thus if we can solve for the unknowns in the form g_f we have $\sum_{j=1}^{T} (n_{i_j}) - 1 \leq \sum_{j=2}^{T} (U_{i_j i_1} + V_{i_j i_1} - 1)$. Thus $S = \{1, 2, \ldots, T\}$ satisfies the condition. □

Using Theorem 3.27 we can compute a bound on the number of evaluation points we need in order to solve for all the unknowns in the form of the GCD.

Theorem 3.28. *If the requirement in Theorem 3.27 is satisfied, that is if there exist $S = \{i_1, i_2, \ldots, i_k\} \subseteq \{1, 2, \ldots, T\}$ such that all i_j's are distinct ($\forall\ 1 \leq j \leq k$) and*

$$(\sum_{j=1}^{k} n_{i_j}) - 1 \leq \sum_{j=2}^{k} (U_{i_j i_1} + V_{i_j i_1} - 1),$$

then using $2n_{max} - 1$ evaluation points, we can solve for all the unknowns.

Proof. The proof is straight forward. Suppose for $S = \{i_1, i_2, \ldots, i_k\}$ the requirement is satisfied. Let $\beta_J = U_{Ji_1} + V_{Ji_1} - 1$ for $J \in S - \{i_1\}$. Without loss of generality assume $\beta_{i_1} \leq \beta_{i_2} \leq \cdots \leq \beta_{i_k}$. Using N images, the J'th coefficient will contribute $\min(\beta_J, N)$ independent evaluations, hence the total number of equations would be

$$\min(\beta_{i_1}, N) + \min(\beta_{i_2}, N) + \ldots, \min(\beta_{i_k}, N)$$

hence For $N = \beta_{i_k}$, the number of equations will be

$$\beta_{i_1} + \beta_{i_2} + \cdots + \beta_{i_k} = \sum_{j=2}^{k} (U_{i_j i_1} + V_{i_j i_1} - 1)$$

which is greater than or equal to the number of unknowns which is $\sum_{j=1}^{k} (n_{i_j}) - 1$ hence using β_k evaluation points we can solve for the unknowns in the coefficients of g_f with indices in S. Hence the total number of evaluation points needed is $\max(\beta_k, n_{max})$. There are two cases. First assume $\beta_k > n_{i_1} + n_k - 1$. In this case, we can use only the i_1'th and k'th coefficient of the GCD to solve for the unknowns in C_{i_1} and then scale the univariate images based on the image of C_{i_1} and solve the rest of the linear systems independently, hence the total number of images needed would be $\max(n_{i_1} + n_k - 1, n_{max}) \leq 2n_{max} - 1$. Now if $\beta_k \geq n_{i+1} + n_k - 1$, then $\beta_k \leq 2n_{max} - 1$, hence the total number of evaluations needed is $\max(\beta_k, n_{max}) \leq 2n_{max} - 1$. □

The proof of Theorem 3.28 is constructive. The following is an example of our new method.

Example 3.29. Let $p = 17$ be the prime and $g = (y^2+1)x^2-(y^3+y)x+(y^3-2y+7)$ be the same GCD as in Example 3.23. Here $n_1 = n_2 = 2$ and $n_3 = 3$. We have $\frac{C_2}{C_1} = y$ so $U_{21} = 1$ and $V_{21} = 1$. We have $n_1+n_2 = 4 > U_{12}+V_{12}$ so we can not use C_1 and C_2 to solve for the leading coefficient. But $\frac{C_3}{C_1} = \frac{y^3-2y+7}{y^2+1}$ so $U_{31} = 3$ and $V_{31} = 2$ hence $n_1+n_3 = 5 \leq U_{31}+V_{31}$.

Thus we can scale the univariate images based on the first coefficient and use the images of the third coefficient to solve for the unknowns in C_1 and C_3. The form for the GCD is $g_f = (A_1y^2 + A_2)x^2 + (A_3y^3 + A_4y)x + (A_5y^3 + A_6y + A_7)$. We need $n_1 + n_3 - 1 = 4$ equations. Suppose we choose the evaluation points $y = 1, y = 7, y = 14$ and $y = 15$. The four univariate images are

$$\begin{aligned} g_1 &= x^2 + 16x + 3, \\ g_2 &= x^2 + 10x + 4, \\ g_3 &= x^2 + 3x + 2, \\ g_4 &= x^2 + 2x + 4. \end{aligned}$$

We fix $A_1 = 1$. Using the third integer coefficients in the above images we get the following set of equations

$$\begin{aligned} \frac{A_5 + A_6 + A_7}{1 + A_2} &= 3 \Rightarrow A_5 + A_6 + A_7 - 3A_2 = 3, \\ \frac{3A_5 + 7A_6 + A_7}{1 + 15A_2} &= 4 \Rightarrow 3A_5 + 7A_6 + A_7 - 9A_2 = 4, \\ \frac{7A_5 + 14A_6 + A_7}{1 + 9A_2} &= 2 \Rightarrow 7A_5 + 14A_6 + A_7 - A_2 = 2, \\ \frac{9A_5 + 15A_6 + A_7}{1 + 4A_2} &= 4 \Rightarrow 9A_5 + 15A_6 + A_7 - 16A_2 = 4. \end{aligned}$$

After solving the above system of linear equations we obtain:

$$\{A_2 = 1, A_5 = 1, A_6 = 15, A_7 = 7\},$$

hence the leading coefficient is $C_1 = y^2 + 1$. We scale the univariate images based on the leading coefficient C_1 evaluated at the corresponding evaluation points to obtain

$$\begin{aligned} g_1' &= g_1 \times (1 + 1) = 2x^2 + 15x + 6, \\ g_2' &= g_2 \times (7^2 + 1) = 16x^2 + 7x + 13, \\ g_3' &= g_3 \times (14^2 + 1) = 10x^2 + 13x + 3, \\ g_4' &= g_4 \times (15^2 + 1) = 5x^2 + 10x + 3. \end{aligned}$$

To solve for the unknowns in C_2 we get the following equations

$$\begin{aligned} A_3 + A_4 &= 15, \\ 3A_3 + 7A_4 &= 7, \\ 7A_3 + 14A_4 &= 13, \\ 9A_3 + 15A_4 &= 10. \end{aligned}$$

Solving the above system results in

$$\{A_3 = A_4 = 16\}$$

hence the GCD is

$$\left(y^2+1\right)x^2 + 16\,\left(y^3+y\right)x + y^3 + 15\,y + 7$$

and we are done.

Now we can use Zippel's idea in [85] to choose the evaluation points carefully so that each system of linear equations will be a transposed Vandermonde system. One can invert a Vandermonde matrix in $O(n^2)$ time and $O(n)$ space compared to $O(n^3)$ time and $O(n^2)$ space for a general matrix. Let $\alpha = (\alpha_2, \alpha_3, \dots, \alpha_n)$ where $\alpha_i \in \mathbb{Z}_p$. Let $g \in F[x_1, \dots, x_n]$ be the GCD that we wish to compute. We choose

$$\alpha^i = (x_2 = \alpha_2^i, x_3 = \alpha_3^i, \dots, x_n = \alpha_n^i)$$

as the $(i+1)$'th evaluation point, i.e. the first evaluation point is $\alpha^0 = (x_2 = 1, \dots, x_n = 1)$, the second evaluation point is $\alpha^1 = (x_2 = \alpha_2, \dots, x_n = \alpha_n)$ and so on. Assume that we have scaled the univariate images of the GCD based on the j'th coefficient. Using this idea, after solving the first system of equations to compute the unknowns for C_j and multiplying univariate images by the image of C_j at the corresponding evaluation point, the rest of linear systems will be transposed Vandermonde systems. We can solve them in quadratic time, linear space and in *parallel*.

Example 3.30. Let $p = 17$ and $g = (yu^3 + 14y^2u^2)x^3 + (3y^2 + 4u - 8)x^2 + (12y^3u^3 - 7y^2u + 14u - 3)x + (2u - v)$. We have $g_f = (A_1yu^3 + A_2y^2u^2)x^3 + (A_3y^2 + A_4u + A_5)x^2 + (A_6y^3u^3 + A_7y^2u + A_8u + A_9)x + (A_{10}u + A_{11}v)$. If we scale the images based on the leading coefficient $yu^3 + 14y^2u^2$ then we can use the fourth coefficient C_4 and solve for the unknowns in C_1 and C_2 using three equations. Since $n_{max} = n_3 = 4$ we need at least four evaluation points. Suppose we choose the evaluation points to be $\alpha^0 = (y = 1, u = 1), \alpha^1 = (y = 2, u = 3), \alpha^2 = (y = 4, u = 9)$ and $\alpha^4 = (y = 8, u = 3^3 \bmod 17 = 10)$.

Suppose we have computed $C_1 = yu^3 + 14y^2u^2$ and $C_4 = 2u - v$. The system of linear equations for C_2 and C_3 are

$$\begin{bmatrix} 1 & 1 & 1 \\ 4 & 3 & 1 \\ 16 & 9 & 1 \end{bmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ a & b & c \\ a^2 & b^2 & c^2 \end{pmatrix}$$

and

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 12 & 12 & 3 & 1 \\ 8 & 8 & 9 & 1 \\ 11 & 11 & 10 & 1 \end{bmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ a & b & c & d \\ a^2 & b^2 & c^2 & d^2 \\ a^3 & b^3 & c^3 & d^3 \end{pmatrix}$$

respectively.

A Problem

Suppose the univariate images of the GCD are scaled based on the i'th coefficient C_j. The evaluation point α^i is said to be *bad*, if $C_j(\alpha^i) = 0$. If α^i is bad, then we can not scale the univariate image of the GCD based on C_j.

Example 3.31. Let $g = (uy^4 + 10uy^2 + 9u)x^2 + (14uy^3 + 2uy + 11y + 4)x + 6u^3 + 4uy$ and $p = 17$. Suppose we are scaling the univariate images based on C_2. We can not use the evaluation point $\alpha^1 = (y = 2, u = 3)$ because $C_2(\alpha^1) \bmod 17 = 0$. The univariate image of the GCD evaluated at α^1 is $g_1 = x^2 + 2$.

We must avoid using a bad evaluation point. But unfortunately, unless we are scaling based on the leading coefficient (or the trailing coefficient), detecting a bad evaluation point beforehand can not be done easily. If we encounter a bad evaluation point we can either scale the univariate images based on a different coefficient or we can throw away this evaluation point and start with a new one. We do the latter.

Remark 3.32. Based on Theorem 3.28, if we can get enough independent equations to solve for all the unknowns in g_f, then having $2n_{max} - 1$ equations is sufficient to have determined systems of linear equations. Thus in our algorithm if using $2n_{max} - 1$ equations does not lead to a determined system of linear equations, we assume that a previous choice of prime or evaluation point is unlucky and restart the algorithms with a new prime or evaluation points.

Benchmarks

We have implemented SparseModGcd algorithm both with and without the optimizations discussed in this chapter in Maple 13. We will compare the performances of these algorithms on two sets of benchmarks given in [30]. The polynomials in the first set have sparse GCDs while the GCDs of the polynomials in the second set are completely dense. All timings are in CPU seconds and were obtained using Maple 13 on a 64 bit Intel Core i7 920 @ 2.66GHz, running Linux, using 31.5 bit primes.

SPARSE-1

Let $m(z) = z^3 - (s+r)z^2 - (t+v)^2 z - 5 - 3u$. For $n = 1, 2, \ldots, 10$, let $f_1 = a \times g$ and $f_2 = b \times g$ where

$$g = sx_1^n + tx_2^n + ux_3^n + \sum_{j=1}^{4}\sum_{i=0}^{n-1} r_{i_j}^{(1)} z^{j-1} x_j^i + \sum_{w=[r,s,t,u,v]} \sum_{k=0}^{n} r_{w_k}^{(1)} w^k,$$

$$a = tx_1^n + ux_2^n + sx_3^n + \sum_{j=1}^{4}\sum_{i=0}^{n-1} r_{i_j}^{(2)} z^{j-1} x_j^i + \sum_{w=[r,s,t,u,v]} \sum_{k=0}^{n} r_{w_k}^{(2)} w^k,$$

$$b = ux_1^n + sx_2^n + tx_3^n + \sum_{j=1}^{4}\sum_{i=0}^{n-1} r_{i_j}^{(3)} z^{j-1} x_j^i + \sum_{w=[r,s,t,u,v]} \sum_{k=0}^{n} r_{w_k}^{(3)} w^k$$

and each $r_{j_k}^{(i)}$ is a positive random integer less than 100. Thus we have 10 GCD problems, all with one field extension $m(z)$, five parameters r, s, t, u and v and four variables x_1, x_2, x_3 and x_4. Each input polynomial is of degree $2n$ in the first three variables and $2n-2$ in x_4. We wanted a data set of polynomials which are sparse but not too sparse. Also the GCD g is sparse (g has $9n+3$ terms and $\deg(g) = n$ in any of x_1, x_2 and x_3). The timings are given in Table 3.2.

Table 3.2: Timings (in CPU seconds) for SPARSE-1

n	SparseModGcd	Optimized SparseModGcd	Speedup (percentage)
10	12.087	10.975	10.13%
15	39.315	34.879	12.71%
20	95.554	85.683	11.52%
25	196.126	176.651	11.02%
30	368.859	329.680	11.88%
35	641.733	571.149	12.35%
40	1007.663	938.878	7.32%
45	1528.628	1426.273	7.17%
50	2278.426	2178.779	4.57%

DENSE-1

Let $m(z) = z^2 - sz - 3$. Suppose g, a and b are three randomly chosen polynomials in x_1, x_2, s and z of total degree n which are dense. That is, the term $x_1^{d_1} x_2^{d_2} s^{d_3} z^{d_4}$ with $0 \leq d_1 + d_2 + d_3 + d_4 \leq n$ is present in each of these three polynomials. So each has exactly $\sum_{i=0}^{n} \binom{i+4}{4}$ terms before we reduce by $m(z)$. For $n = 5, 6, \ldots, 10, 15, 20, 30$, let $f_1 = g \times a$ and $f_2 = g \times b$. The timings are given in Table 3.3.

Table 3.3: Timings (in CPU seconds) for DENSE-1

n	SparseModGcd	Optimized SparseModGcd	Speedup (percentage)
5	0.202	0.147	37.41%
6	0.360	0.227	58.59%
7	0.584	0.359	62.39%
8	0.861	0.553	57.14%
9	1.326	0.747	77.51%
10	2.297	1.398	64.30%
15	13.311	7.098	87.53%
20	60.029	27.927	114.94%
30	521.690	239.638	117.70%

The data in Tables 3.2 and 3.3 show that our optimized algorithm has a better performance compared to the old algorithm especially for dense polynomials. This is because for dense polynomials, the bottleneck of the algorithm is trial divisions in both zero and non-zero characteristic.

Chapter 4

Factorization

In a computer algebra system, computations with polynomials over algebraic function fields such as computing GCDs and factorization arise, for example, when one solves non-linear polynomial equations involving parameters.

There are various algorithms for factoring polynomials over an algebraic number or function field L. We discussed some of these algorithms in Section 1.3.3.

Our main focus in this chapter is factoring *multivariate* polynomials in $L[x_1, \ldots, x_v]$ using Hensel lifting. We evaluate all parameters and all variables except one at small integers thus reducing the factorization in $L[x_1, \ldots, x_v]$ to univariate factorization in x_1 over a number ring. There are a number of efficient algorithms for univariate factorization [67, 47, 52]. We choose to use Trager's algorithm [67] because it is highly effective in implementation for the fields and degrees we are interested in. We choose the evaluation point $\alpha \in \mathbb{Z}^k$ such that $L(\alpha)$, the number ring obtained by evaluating the parameters $t_1, \ldots, t_k$ in L at α, is a field.

When one uses Hensel lifting, a challenge is to solve the leading coefficient problem for lifting non-monic polynomials. Abbott in [1], suggests using a trick by Kaltofen in [38] which recursively computes the leading coefficients from their bivariate images using Hensel lifting. Our approach is to modify Wang's ingenious method given in [77] for factoring polynomials over $\mathbb{Z}$. Let $f \in \mathbb{Z}[x_1, \ldots, x_v]$. His idea is to first factor the leading coefficient $l(x_2, \ldots, x_v) = \mathrm{lc}_{x_1}(f)$ of the input polynomial f in the main variable x_1, recursively. Then evaluate all the variables except x_1 at an evaluation point $\alpha \in \mathbb{Z}^{v-1}$ and factor the univariate polynomial $f(x_1, \alpha)$. Now using the integer leading coefficients of the univariate factors, one can determine which factor of $l(x_2, \ldots, x_v)$ belongs to the leading coefficient of which factor of $f(\alpha)$. To do this, Wang identifies unique prime divisors for each factor of $l(x_2, \ldots, x_v)$ evaluated at α by computing integer GCDs only. Unfortunately this idea does not generalize to L. We show an example.

Example 4.1. Let $L = \mathbb{Q}(\sqrt{-5})$ and

$$f = ((y + \sqrt{-5} + 1)x + 1)((y + \sqrt{-5} - 1)x + 1)$$

$$= (y^2 + 2\sqrt{-5}y - 6)x^2 + 2(y + \sqrt{-5})x + 1.$$

We have $\mathrm{lc}_x(f) = y^2 + 2\sqrt{-5}y - 6 \in L[y]$, so if we evaluate y at $\alpha \in \mathbb{Z}$, we will obtain an element of $\mathbb{Z}[\sqrt{-5}]$. But $\mathbb{Z}[\sqrt{-5}]$ is not a unique factorization domain thus GCDs do not always exist in this ring. For example, for $y = 0$ we have $\mathrm{lc}_x(f)(y = 0) = -6 = -2 \times 3 = -(1 - \sqrt{-5}) \times (1 + \sqrt{-5})$.

An easy solution to the leading coefficient problem is to make the input polynomial monic by shifting the variables $x_2, \ldots, x_v$. We illustrate with an example.

Example 4.2. In Example 4.1, if we substitute $y = x + 2y$ in f, we will obtain

$$\tilde{f} = f(x, x + 2y) = (x^2 + 2xy + \sqrt{-5}x + x + 1)(x^2 + 2xy + \sqrt{-5}x - x + 1) =$$

$$x^4 + 4\,yx^3 - 4\,x^2 + 4\,y^2x^2 + 4\,yx + \left(2\,x^3 + 4\,yx^2 + 2\,x\right)\sqrt{-5} + 1.$$

We have $\mathrm{lc}_x(\tilde{f}) = 1$. One can obtain factors of f by substituting $y = x - 2y$ in the factors of $\tilde{f}$. The problem with this solution is that it could make the input polynomial f denser.

Another problem is that one needs to do computations with fractions in Hensel lifting. To solve this problem, one can work modulo p^l, a power of a prime. This modulus, p^l, must be at least twice the largest integer coefficient in any factor of f. Unfortunately the known bounds on the sizes of the integer coefficients in the factors of f are usually very big which makes the computations really slow. In [1] it is suggested that it is better not to do the calculations modulo p^l because of the bad bounds but instead to lift over $\mathbb{Q}$. In our algorithm we choose a prime p of a modest size and then lift the integer coefficients to their correct values using a new multivariate p-adic lifting algorithm which uses a sparse interpolation method similar to Zippel's algorithm [84].

This chapter is organized as follows. In Section 4.1 we present an example showing the main flow and the key features of our algorithm. We then identify possible problems that can occur and how the new algorithm deals with them in Section 4.2. In Section 4.3 we present our new algorithm. Finally, in Section 4.4 we compare Maple implementations of our algorithm with Trager's algorithm for a set of polynomials.

4.1 An Example

Let $F = \mathbb{Q}(t_1, \ldots, t_k)$, $k \geq 0$. For i, $1 \leq i \leq r$, let $m_i(z_1, \ldots, z_i) \in F[z_1, \ldots, z_i]$ be monic and irreducible over $F[z_1, \ldots, z_{i-1}]/ \langle m_1, \ldots, m_{i-1} \rangle$. Let $L = F[z_1, \ldots, z_r]/ \langle m_1, \ldots, m_r \rangle$. L is an algebraic function field in k parameters $t_1, \ldots, t_k$ (this also includes number fields). Let f be a non-zero square-free polynomial in $L[x_1, \ldots, x_v]$. Our problem is given f, compute $f_1, f_2, \ldots, f_n$ such that $f = \mathrm{lc}_{x_1,\ldots,x_v}(f) \times f_1 \times f_2 \times \cdots \times f_n$ where f_i is a monic irreducible polynomial in $L[x_1, \ldots, x_v]$.

Our algorithm works with the *monic associate* $\tilde{f}$ of the input f and *primitive associates* of the minimal polynomials which we now define.

Definition 4.3. Let $D = \mathbb{Z}[t_1, \ldots, t_k]$. A non-zero polynomial in $D[z_1, \ldots, z_r, x_1, \ldots, x_v]$ is said to be *primitive* wrt $(z_1, \ldots, z_r, x_1, \ldots, x_v)$ if the GCD of its coefficients in D is 1. Let f be non-zero in $L[x_1, \ldots, x_v]$. The denominator of f is the polynomial $den(f) \in D$ of least total degree in $(t_1, \ldots, t_k)$ and with smallest integer content such that $den(f)f$ is in $D[z_1, \ldots, z_r, x_1, \ldots, x_v]$. The primitive associate, $\check{f} = \mathrm{prim}(f)$, of f is the associate of $den(f)f$ which is primitive in $D[z_1, \ldots, z_r, x_1, \ldots, x_v]$ and has positive leading coefficient in a term ordering. The *monic associate* $\tilde{f}$ of f is defined as $\tilde{f} = \mathrm{prim}(\mathrm{monic}(f))$. Here $\mathrm{monic}(f)$ is defined by $\mathrm{lc}_{x_1,\ldots,x_v}(f)^{-1} f$.

Example 4.4. Let $f = 3tx^2 + 6tx/(t^2 - 1) + 30tz/(1 - t)$ where $m_1(z) = z^2 - t$. Here $f \in L[x]$ where $L = \mathbb{Q}(t)[z]/ \left\langle z^2 - t \right\rangle$ is an algebraic function field in one parameter t. We have $\mathrm{den}(f) = t^2 - 1$ and $\check{f} = \mathrm{prim}(f) = \mathrm{den}(f)f/(3t) = (t^2 - 1)x + 2x - 10z(t + 1)$. For $f = 2zx^2 + 2/t$ we have $\mathrm{prim}(f) = tzx^2 + 1$, $\mathrm{monic}(f) = x^2 + z/t^2$ and $\tilde{f} = t^2x^2 + z$.

We demonstrate our algorithm using the following example using t for a parameter and x and y for variables.

Example 4.5. Let $m(z) = z^2 - t^3 + t$ and

$$f = (t^3 - t)y^2x^2 + (20t^3z - t^2z - 20tz + z)yx^2 +$$

$$(-20t^5 + 40t^3 - 20t)x^2 + (-tz + 21z)yx + (421t^3 - 421t)x - 21t$$

$$= \frac{1}{t^2-1} \times \tilde{f}_1 \times \tilde{f}_2 = \frac{1}{t^2-1}((t^2-1)xy + (t^2-1)20zx - z)((t^3-t)xy - (t^2-1)zx + 21z)$$

$$= (t^3-t)(xy + 20zx - \frac{z}{t^2-1})(xy - \frac{zx}{t} + \frac{21z}{t^3-t}).$$

Here $L = \mathbb{Q}(t)[z]/\left\langle z^2 - t^3 + t \right\rangle$ and $f \in L[x, y]$. We have $\text{prim}(f) = f$ and $\tilde{m} = m$. The first step in our algorithm is to eliminate any algebraic elements in $\gamma = \text{lc}_{x,y}(\text{prim}(f)) = t^3 - t$ by computing $\tilde{f}$. This is done to avoid any fractions in the parameter t in the Hensel lifting. Since γ does not involve the algebraic element z, we have $\tilde{f} = \text{prim}(f)$.

Suppose we choose x as the main variable. In order to use Hensel lifting we factor the leading coefficient

$$\text{lc}_x(\tilde{f}) = (t^3-t)y^2 + (20t^3z - t^2z - 20tz + z)y - 20t^5 + 40t^3 - 20t.$$

We do this by recursively using our algorithm in one less variable. The base case, for univariate polynomials over a number field, is done by using Trager's algorithm [67]. We will obtain

$$\text{lc}_x(\tilde{f}) = \gamma \times l_1 \times l_2 = (t^3-t)(y - z/t)(y + 20z).$$

Now we clear the denominators in l_i's to obtain $\text{lc}_x(\tilde{f}) = \bar{\gamma} \times \tilde{l}_1 \times \tilde{l}_2 = (t^2-1)(ty-z)(y+20z)$. In order to factor $\tilde{f}$, we evaluate it at a point α for all the parameters and variables except the main variable, x. We factor the resulting univariate polynomial in $\mathbb{Q}[z][x]/\left\langle \tilde{m}(\alpha) \right\rangle$ using Trager's algorithm and then we lift the variables and parameters one by one using Hensel lifting. Suppose we choose the evaluation point to be $\alpha = (t = 12, y = 5)$. This evaluation point must satisfy certain conditions that we will discuss in Section 4.2.2. We have

$$\tilde{f}(\alpha) = (170885z - 4864860)\, x^2 + (45z + 722436)\, x - 252$$

and $\tilde{m}(\alpha) = z^2 - 1716$ which is irreducible hence $L(\alpha)$ is a field, one condition on α. Using Trager's algorithm, we factor $\tilde{f}(\alpha)$ over $L(\alpha)$ to get $\tilde{f}(\alpha) = \text{lc}_x(\tilde{f}(\alpha)) \times u_1 \times u_2$ with u_1, u_2 monic. We obtain

$$\tilde{f}(\alpha) = (170885z - 4864860) \times (x + \frac{1}{19630325}z - \frac{48}{137275}) \times (x + \frac{105}{22451}z + \frac{21}{157}).$$

We next factor $\bar{\gamma} = t^2 - 1$ over $\mathbb{Z}$ to obtain $\bar{\gamma} = t^2 - 1 = \tilde{l}_3 \times \tilde{l}_4 = (t-1)(t+1)$. Before doing Hensel lifting, we determine the true leading coefficient of each factor of $\tilde{f}$ in $\mathbb{Z}[t][y]$. To do this, we use the denominators of u_1 and u_2. We know that

$$d_i = \text{den}(u_i) \mid \text{den}(\frac{1}{\text{lc}_x(\tilde{f}_i(\alpha))})$$

where $\tilde{f}_i$ is a factor of $\tilde{f}$. We have

$$\begin{aligned}
d_1 &= \mathrm{den}(u_1) = 19630325 = (5)^2(11)(13)(17)^2(19),\\
d_2 &= \mathrm{den}(u_2) = 22451 = (11)(13)(157),\\
D_1 &= \mathrm{den}(1/\tilde{l}_1(\alpha)) = 1884 = (2)^3(3)(157) \text{ where } \tilde{l}_1 = ty - z,\\
D_2 &= \mathrm{den}(1/\tilde{l}_2(\alpha)) = (5)^2(17)^2(19) \text{ where } \tilde{l}_2 = y + 20z,\\
D_3 &= \mathrm{den}(1/\tilde{l}_3(\alpha)) = 11 \text{ where } \tilde{l}_3 = t - 1,\\
D_4 &= \mathrm{den}(1/\tilde{l}_4(\alpha)) = 13 \text{ where } \tilde{l}_4 = t + 1.
\end{aligned}$$

The evaluation point α was chosen so that D_i's have a set of distinct prime divisors, namely $\{3, 17, 11, 13\}$. Here D_i's are relatively prime so we have

$$\gcd(d_i, D_j) > 1 \Rightarrow \tilde{l}_j \mid \bar{l}_i$$

where $\bar{l}_i = \mathrm{lc}_{x_1}(\tilde{f}_i)$. Using this we obtain $\bar{l}_1 = (t^2-1)(y+20z)$ and $\bar{l}_2 = (t^2-1)(ty-z)$ and we have

$$\tilde{f} \equiv \frac{1}{t^2-1} \times (\bar{l}_1(\alpha)u_1) \times (\bar{l}_2(\alpha)u_2) \bmod \langle t-12, y-5\rangle .$$

To avoid fractions in $\mathbb{Q}(t)$ in the Hensel lifting we multiply

$$\tilde{f} := \frac{\bar{l}_1 \times \bar{l}_2}{\mathrm{lc}_{x_1}(\tilde{f})} \times \tilde{f} = (t^2-1) \times \tilde{f}$$

and use this new $\tilde{f}$ in Hensel lifting. Now we use Hensel lifting to lift the parameter t and the variable y in the other coefficients of the $\tilde{f}_i$. To avoid any computations with fractions in $\mathbb{Q}$, we do the calculations modulo a prime, say $p = 17$. After applying Hensel lifting we obtain the factors $\bar{f}_1 = ((t^2-1)(y+20z)x - z)$ and $\bar{f}_2 = ((t^2-1)(ty-z)x + 4z)$ s.t. $\tilde{f} \equiv \bar{f}_1 \times \bar{f}_2 \pmod{17}$. The final task is to find the integer coefficients of $\bar{f}_1$ and $\bar{f}_2$ so that $\tilde{f} = \bar{f}_1 \times \bar{f}_2$. To do this, we use sparse interpolation. We have $e_1 = \tilde{f} - \bar{f}_1 \times \bar{f}_2 \bmod \langle \tilde{m}\rangle$, the first error polynomial over $\mathbb{Z}$. We want to find $\sigma_1, \sigma_2 \in L[x, y]$ such that

$$\tilde{f} \equiv (\bar{f}_1 + \sigma_1 \times p)(\bar{f}_2 + \sigma_2 \times p) \bmod p^2.$$

Assuming that our choice of α and p has not caused any terms in the polynomials $\bar{f}_1$ and $\bar{f}_2$ to vanish, we know that the monomials in σ_1 and σ_2 are the same as those in $\bar{f}_1$ and $\bar{f}_2$ respectively, so we have the *assumed forms* for σ_1 and σ_2. If p is large and α is chosen randomly, this would happen with high probability. Since $\bar{f}_1$ and $\bar{f}_2$ have correct leading coefficients we have $\sigma_1 = Az$ and $\sigma_2 = Bz$ for unknown integer coefficients A and B. To find the values for A and B we have

$$(\sigma_1 \times \bar{f}_2 + \sigma_2 \times \bar{f}_1) \bmod \langle \tilde{m} \rangle - \frac{e_1}{p} \equiv 0 \bmod p.$$

After equating every coefficient in monomials in x, y, z and t in the above expression to zero, we get the following linear system:

$$\{A = 0, -B + 1 = 0, B - 1 = 0, -1 - 4\,A + B = 0, 1 - B + 4 = 0,$$

$$A = 0, -A - 20 + 20\,B = 0, 2\,A + 40 - 40\,B = 0, -A = 0\}.$$

Solving modulo p, we get $A = 0$ and $B = 1$ so we update

$$\bar{f}_1 := \bar{f}_1 + \sigma_1 \times p = ((t^2 - 1)(y + 20z)x - z)$$

and

$$\bar{f}_2 := \bar{f}_2 + \sigma_2 \times p = ((t^2 - 1)(ty - z)x + 21z).$$

Now we have $\tilde{f} \equiv \bar{f}_1 \times \bar{f}_2 \bmod p^2$. This time the new error $e_2 = \tilde{f} - \bar{f}_1 \times \bar{f}_2 \bmod \langle \tilde{m} \rangle$ is zero, so we have

$$\tilde{f} = \bar{f}_1 \times \bar{f}_2 = ((t^2 - 1)xy + (t^2 - 1)20zx - z)((t^3 - t)xy - (t^2 - 1)zx + 21z).$$

To complete the factorization of f we have $f = \mathrm{lc}_{x,y}(f) \times \mathrm{monic}(\bar{f}_1) \times \mathrm{monic}(\bar{f}_2)$, thus

$$f = (t^3 - t)(xy + 20zx - \frac{z}{t^2 - 1})(xy - \frac{zx}{t} + \frac{21z}{t^3 - t})$$

and we are done.

4.2 Problems

In the example we mentioned that the evaluation point must satisfy certain conditions in order for the algorithm to work properly. Another issue is the defect of the algebraic function field L which is the biggest denominator of any algebraic integer in L (See [1, 81]). Here we identify all problems.

4.2.1 The Defect

Unlike factorization over $\mathbb{Q}$, when factoring a polynomial $\tilde{f}$ over the algebraic field L, the leading coefficient of a factor $\tilde{f}_i$ in the variables $x_1, \ldots, x_v$ might not divide the leading coefficient of $\tilde{f}$, i.e. $\mathrm{lc}_{x_1,\ldots,x_v}(\tilde{f}_i) \nmid \mathrm{lc}_{x_1,\ldots,x_v}(\tilde{f})$ in $\mathbb{Z}[t_1, \ldots, t_k]$.

Example 4.6. Let $m = z^2 - t^3$, $L = \mathbb{Q}(t)[z]/\langle m \rangle$ and $f = x^2 - t$. We have $\tilde{f} = f$ and

$$f = (x - \frac{z}{t})(x + \frac{z}{t}) = \frac{1}{t^2}(tx - z)(tx + z).$$

Here $\tilde{f}_1 = tx - z$ but $\mathrm{lc}_x(\tilde{f}_1) = t \nmid \mathrm{lc}_x(\tilde{f}) = 1$.

The denominator t in this example is a divisor of the *defect* of the algebraic function field L. The defect is the largest rational integer that divides the denominator of any algebraic integer in the field (See e.g. [6]).

Theorem 4.7 (See Abbott [1])**.** Let $\Delta \in \mathbb{Z}[t_1, \ldots, t_k]$ be the discriminant of the algebraic field L. The *defect* is the biggest square that divides Δ.

Example 4.8. When $r = 1$ (one field extension), $\Delta = \mathrm{res}_{z_1}(M, M')$ where $M = \tilde{m}_1$. For example, for $\tilde{m}_1 = z_1^2 - t^3$ we have $\Delta = \mathrm{res}_{z_1}(z_1^2 - t^3, 2z_1) = -4t^3$ and hence $2t$ is the defect.

Theorem 4.9 (See Abbott [1])**.** Let $d_i = \deg_{z_i}(m_i)$. The discriminant of L is

$$\Delta = \prod_{i=1}^{r} N_1(N_2(\ldots(N_{i-1}(\mathrm{discr}(\tilde{m}_i)^{d_{i+1}\cdots d_r}))\ldots))$$

where $N_i(f) = \mathrm{res}_{z_i}(f, \tilde{m}_i)$ and $\mathrm{discr}(\tilde{m}_i) = \mathrm{res}_{z_i}(M_i, M_i')$ where $M_i = \tilde{m}_i$.

Suppose using Theorem 4.9 we have computed the discriminant $\Delta \in \mathbb{Z}[t_1, \ldots, t_k]$. Let $\delta \times D_1^{e_1} \times \cdots \times D_k^{e_k}$ be a square-free factorization of Δ where $\delta \in \mathbb{Z}$. Since we want to avoid integer factorization, we choose $\mathbb{D}$ to be this integer multiple of the defect:

$$\mathbb{D} = \delta \times D_1^{\lfloor \frac{e_1}{2} \rfloor} \times \cdots \times D_k^{\lfloor \frac{e_k}{2} \rfloor}.$$

Theorem 4.10 (See [81])**.** If $\tilde{f}_i$ is a factor of $\tilde{f}$ and $\mathbb{D}$ is an integral multiple of the defect, then

$$\mathrm{lc}_{x_1,\ldots,x_v}(\tilde{f}_i) \mid \mathbb{D} \times \mathrm{lc}_{x_1,\ldots,x_v}(\tilde{f}).$$

Remark 4.11. In this chapter, we always assume the lexicographic monomial ordering with $x_1 > x_2 > \cdots > x_v$.

Remark 4.12. The leading coefficient of $\tilde{f}$, $\mathrm{lc}_{x_1,\ldots,x_v}(\tilde{f}) \in \mathbb{Z}[t_1,\ldots,t_k]$, may not *split* among the leading coefficients of the factors. That is $\prod_{i=1}^{n} \mathrm{lc}_{x_1,\ldots,x_v}(\tilde{f}_i)$ may not divide $\mathbb{D}^l \times \mathrm{lc}_{x_1,\ldots,x_v}(\tilde{f})$ for any $l \in \mathbb{Z}^+$.

Example 4.13. In Example 4.5, the discriminant of L is $-4t^3+4t$ and hence $\mathbb{D}=4$. We have $\mathrm{lc}_x\tilde{f} = t^3-t, \mathrm{lc}_x\tilde{f}_1 = t^2-1$ and $\mathrm{lc}_x\tilde{f}_2 = t^3-t$. We have $(t^2-1)\times(t^3-t) \nmid \mathbb{D}^l \times (t^3-t)$ for any $l \in \mathbb{Z}^+$.

4.2.2 Good and Lucky Evaluation Points

Definition 4.14 (Good Evaluation Points)**.**

Let $\alpha = (t_1=\alpha_1,\ldots,t_k=\alpha_k, x_2=\beta_2,\ldots,x_v=\beta_v) \in \mathbb{Z}^{k+v-1}$ be the evaluation point that we choose in our algorithm to factor the univariate polynomial $\tilde{f}(\alpha)$. We impose the following conditions on α. We say α is good if:

1. The leading coefficient of $\tilde{f}$ in the main variable x_1 and the leading coefficient of $\tilde{m}_i$ in z_i do not vanish after evaluating at α, i.e. $\deg_{x_1}(\tilde{f}) = \deg_{x_1}(\tilde{f}(\alpha))$ and $\deg_{z_i}(\tilde{m}_i) = \deg_{z_i}(\tilde{m}_i(\alpha))$.

2. $L(\alpha)$ remains a field so that we still have unique factorization of $\tilde{f}(\alpha)$. As an example, the evaluation point $t=1$ is not a *good* choice for our Example 4.4 because the minimal polynomial z^2-t evaluated at this point is no longer irreducible.

3. The polynomial $\tilde{f}$ evaluated at α remains square-free in x_1, i.e. $\gcd(\tilde{f}(\alpha), \tilde{f}'(\alpha)) = 1$ in $L(\alpha)[x_1]$, so that we can apply Hensel lifting.

4. The fourth condition on the evaluation point α enables us to distribute factors of $\mathrm{lc}_{x_1}(\tilde{f})$ to the monic univariate factors $u_1,\ldots,u_n$ where $u_i \in L(\alpha)[x_1]$ and

$$\tilde{f}(\alpha) = \mathrm{lc}_{x_1}(\tilde{f})(\alpha) \times u_1 \times \cdots \times u_n.$$

   Suppose $\gamma \times \hat{l}_1^{e_1} \times \cdots \times \hat{l}_m^{e_m}$ is the factorization of $\mathrm{lc}_{x_1}(\tilde{f})$ over L and $\mathbb{D}$ is the defect. Here $\gamma \in \mathbb{Z}[t_1,\ldots,t_k]$ and $\hat{l} \in L[x_2,\ldots,x_n]$. Let $\beta = \mathbb{D} \times \gamma = \Omega \times \beta_1^{c_1} \times \beta_2^{c_2} \times \cdots \times \beta_k^{c_k} \in \mathbb{Z}[t_1,\ldots,t_k]$ where $\Omega \in \mathbb{Z}$ and $\beta_i \in \mathbb{Z}[t_1,\ldots,t_k]$ is irreducible over $\mathbb{Z}$. Let $\bar{d}_i = \mathrm{den}(1/\hat{l}_i(\alpha))$. In order to be able to uniquely distribute the factors of $\mathbb{D} \times \mathrm{lc}_{x_1}(\tilde{f})$ to the univariate factors, we require that numbers in the set

$$A = \{\beta_1(\alpha), \dots, \beta_k(\alpha), \bar{d}_1, \dots, \bar{d}_m\}$$

have distinct prime divisors that do not divide Ω (See Example 4.15 below).

Also, a prime p is considered to be a *good* prime if the leading coefficient of $\tilde{f}$ in x_1 and the leading coefficient of $m_i(\alpha)$ in z_i do not vanish modulo p.

Example 4.15. In Example 4.5 we have $\beta_1 = t-1, \beta_2 = t+1$, $\hat{l}_1 = ty - z, \hat{l}_2 = y + 20z$, $\mathbb{D} = 2$ and $\Omega = 2$. We can not use the evaluation point $\alpha = (t = 3, y = 5)$ because the numbers in $A = \{2 = (2), 4 = (2)^2, 417 = (3)(139), 9551 = (9551)\}$ do not have distinct prime divisors.

Remark 4.16. Let $\mathrm{lc}_{x_1}(\tilde{f}) = \gamma \times \hat{l}_1^{e_1} \times \cdots \times \hat{l}_m^{e_m}$ where $\hat{l}_i$ is an irreducible factor of $\mathrm{lc}_{x_1}(\tilde{f})$. Let $N_i = \mathrm{norm}(\hat{l}_i) \in \mathbb{Q}(t_1, \dots, t_k)[x_1, \dots, x_v]$. If $\exists i \neq j$ such that $N_i \mid N_j$ over $\mathbb{Q}(t_1, \dots, t_k)$ then condition 4 may not be satisfied, no matter what α is. This especially happens if $\hat{l}_i$ and $\hat{l}_j$ are conjugates hence $N_i = N_j$. In this case, the denominators $\bar{d}_i = \mathrm{den}(1/\hat{l}_i(\alpha))$ and $\bar{d}_j = \mathrm{den}(1/\hat{l}_j(\alpha))$ will be images of the same polynomial $\mathrm{norm}(\hat{l}_i) = \mathrm{norm}(\hat{l}_j)$ (See [67]). Here we need to do something else. The simplest solution, as shown in Example 4.2 is to shift the variables $x_2, x_3, \dots, x_v$ in the input polynomial by computing

$$\tilde{f} := \tilde{f}(x_1, x_1 + c_2x_2, x_1 + c_3x_3, \dots, x_1 + c_vx_v)$$

for some randomly chosen $c_i \in \mathbb{Z}$. Now $\mathrm{lc}_{x_1}(\tilde{f}) \in \mathbb{Z}[t_1, \dots, t_k]$. The the leading coefficient of $\tilde{f}$ in x_1 will not involve any of the variables $x_2, x_3, \dots, x_v$. The following is an example.

Example 4.17. Let $\tilde{m} = z^2 - t$ and $\tilde{f} = x^2y^2 - tx^2 + 2txy + t^2 = ((y+z)x+t)((y-z)x+t)$. We have $\mathrm{lc}_x(\tilde{f}) = y^2 - t = \hat{l}_1 \times \hat{l}_2 = (y-z)(y+z)$ and $\mathrm{norm}(\hat{l}_1) = \mathrm{norm}(\hat{l}_2) = y^2 - t$. If we choose $\alpha = (y = 1, t = 6)$ we will have $\bar{d}_1 = \mathrm{den}(1/\hat{l}_1(\alpha)) = 5$ and $\bar{d}_2 = \mathrm{den}(1/\hat{l}_2(\alpha)) = 5$ and the set $A = \{5, 5\}$ will not have a set of distinct prime divisors. If we shift the variable y to $x + 3y$, we will get $\tilde{f} := \tilde{f}(x, x+3y) = (x^2 + (3y+z)x + t)(x^2 + (3y-z)x + t)$ and $\mathrm{lc}_x(\tilde{f}) = 1$.

Definition 4.18 (Lucky Evaluation Point)**.** A good evaluation point $\alpha \in \mathbb{Z}^{v+k-1}$ is said to be *lucky* if it satisfies the following conditions, otherwise it is *unlucky.*

(i) The number of irreducible factors of $\tilde{f}(\alpha)$ over $L(\alpha)$ is the same as the number of irreducible factors of $\tilde{f}$ over L.

(ii) If $\hat{l}_i \mid \mathrm{lc}_{x_1}(\tilde{f}_j)$ where $\tilde{f}_j$ is an irreducible factor of $\tilde{f}$ then $\gcd(\mathrm{den}(1/\hat{l}_i(\alpha)), \mathrm{lc}_{x_1}(\tilde{u}_j)) \neq 1$.

(iii) If $\beta_i \mid \mathrm{lc}_{x_1}(\tilde{f}_j)$ then $\gcd(\beta_i(\alpha), \mathrm{lc}_{x_1}(\tilde{u}_j)) \neq 1$

(iv) α does not *annihilate* any terms of any factor $\tilde{f}_i$ of $\tilde{f}$ (See Example 4.19 below).

Similarly, a good prime p is said to be *lucky* if no terms of any factor $\tilde{f}_i$ of $\tilde{f}$ is *annihilated* modulo p. If the evaluation point α or prime p is unlucky, the algorithm must detect this and restart using a new good evaluation point and a new good prime.

Example 4.19. Let $\tilde{f} = \tilde{f}_1 \times \tilde{f}_2$ where $\tilde{f}_1 = x^2 - (t-15)zx - tz + 1$ and $\tilde{f}_2 = x^3 - 17tzx + 1$ where $z = \sqrt{t-1}$. Here the evaluation point $t = 15$ is good but it is unlucky because it annihilates the term $(t-15)zx$ in $\tilde{f}_1$. Similarly, the prime $p = 17$ is unlucky because the term $17tzx$ in $\tilde{f}_2$ vanishes modulo p. Also, $t = 0$ is unlucky because $\tilde{f}_2(t = 0)$ factors but $\tilde{f}_2$ does not.

Remark 4.20. Since we will use sparse interpolation to lift the integer coefficients of the factors computed using Hensel lifting, the evaluation point α and the prime p must not *annihilate* any terms in any factors of $\tilde{f}$. Unfortunately we will not be able to identify unlucky evaluation points and primes in advance. Instead, if α is unlucky or p is unlucky and the form of any of the correcting polynomials $\sigma_1, \sigma_2, \ldots$ is wrong, the system of linear equations in the sparse interpolation will be inconsistent with high probability. To decrease the probability of choosing an evaluation point (or a prime) that *annihilates* terms in factors of $\tilde{f}$, one should choose α at random from a large set of evaluation points , e.g. $p = 2^{31} - 1$ and $\alpha \in \mathbb{Z}_p$ at random.

Remark 4.21. If α is unlucky and there are extraneous factors in the factorization of $\tilde{f}(\alpha)$ then Hensel lifting will fail with high probability. Hensel lifting may succeed modulo p with low probability if the prime p in Hensel lifting is also unlucky and results in extraneous factors in $\tilde{f} \bmod p$ corresponding to those of $\tilde{f}(\alpha)$.

Example 4.22. Suppose $\tilde{f} = x^2 + 17(t-1)zx - t^2$ and $z = \sqrt{t+1}$. The evaluation point $\alpha = (t = 1)$ is good but unlucky because $\tilde{f}$ is irreducible but $\tilde{f}(\alpha) = (x-1)(x+1)$. If we also chose $p = 17$, Hensel lifting will succeed and return $(x-t)(x+t)$.

If Hensel lifting does not fail when α is unlucky, then we will not be able to lift the integer coefficients of factors of $\tilde{f}$ and the algorithm will restart by choosing a new evaluation point.

4.2.3 Degree Bound for the Parameters

In order to use Hensel lifting, we need to have bounds on the degrees of the parameters and variables in the factors of $\tilde{f}$. For variables $x_1, \ldots, x_v$ we have $\deg_{x_i}(\tilde{f}) = \sum_{j=1}^{n} \deg_{x_i}(\tilde{f}_j)$ but unlike factorization over the rationals, $\deg_{t_i}(\tilde{f}_j)$ is not necessarily bounded by $\deg_{t_i}(\tilde{f})$.

Example 4.23. Let $m = z^2 - \frac{1}{t^3}$ and $\tilde{f} = x^2 - t$. We have

$$\tilde{f} = \tilde{f}_1 \tilde{f}_2 = (x + t^2 z)(x - t^2 z).$$

Here $\deg_t \tilde{f}_1 = \deg_t \tilde{f}_2 = 2 > \deg_t \tilde{f} = 1$.

In [1], Abbott gives a possible bound T_i on the degree of each factor in t_i based on Trager's algorithm which is usually much bigger than the actual degrees of the factors. In our algorithm when we lift the parameter t_i in the factorization of $\tilde{f}$, as soon as the factors have been lifted to the correct degree, the error would be zero (provided the evaluation point and the prime are not unlike) and the algorithm succeeds. However if the evaluation point is unlucky, our algorithm will have to lift the parameter t_i to the degree T_i before realizing it. But the probability that a randomly chosen evaluation point (from a large set of values) is unlucky is low. Thus instead of using the bad bound T_i, we start the algorithm with a heuristic bound T for the degree of the parameters. Now Hensel lifting fails if either the evaluation point is unlucky or the heuristic bound T is not big enough. In this case, we will double the heuristic bound, i.e. $T := 2 \times T$, and restart the algorithm by choosing a new evaluation point. In this way, we will eventually get both a good evaluation point and a big enough bound T and Hensel lifting will eventually succeed.

In our implementation we choose the initial bound T based on the following conjecture from Abbott [1]:

Conjecture 4.24 (Abbott [1])**.**

$$\deg_{t_i}(\tilde{f}_j) \leq \deg_{t_i}(\tilde{f}) + \sum_{l=1}^{r} \deg_{t_i}(\tilde{m}_l).$$

4.2.4 Numerical Bound

Most algorithms that use Hensel lifting (See [82, 1]) either work over $\mathbb{Q}$ or work modulo a power of a prime which must be larger than twice the size of the largest integer coefficients in the factors of $\tilde{f}$. Abbott in [1] presents a bound for this but his bound is very poor. The following is an example from [1].

Example 4.25. Let $\tilde{m} = z^2 - 4t - 1$ and $\tilde{f} = x^2 + x - t = (x + \frac{1+z}{2})(x + \frac{1-z}{2})$. The bound given by Abbott for factoring $\tilde{f}$ is greater than 5000000.

The poor bound means a large modulus is needed which slows Hensel lifting down. Instead, we work modulo a machine prime p and then lift the integer coefficients using our sparse p-adic lifting algorithm if necessary. We still need a bound for the case where α is unlucky and Hensel lifting has not detected this due to the unlucky choice of the prime p (See Example 4.22). For this, we choose a heuristic bound B. Any good estimate for B will work. Now if the sparse p-adic lifting fails, either α is unlucky or p is unlucky or the bound B is not big enough. In this case, we square the bound, i.e. $B := B^2$, and restart the algorithm by using a new evaluation point α and new prime p. In this way, we will eventually get a lucky evaluation point and a lucky prime and a bound big enough to lift the integer coefficients.

4.3 The Algorithm

Algorithm 4.1: **efactor**

```
Require: $f \in L[x_1, x_2, \ldots, x_v]$ where $L$ is the algebraic function field.
Ensure: Factorization of $f$: $f = l \times f_1^{e_1} \times \cdots \times f_n^{e_n}$ where $f_i$ is a monic irreducible polynomial
    and $l = \mathrm{lc}_{x_1,\ldots,x_v}(f)$.
1: If $f \in L$, return $f$.
2: Let $c = \mathrm{cont}_{x_1}(f) \in L[x_2, \ldots, x_n]$ be the content of $f$. If $c \neq 1$ then factor $c$ and $f/c$
    separately using Algorithm efactor and return their product.
3: Do a square-free factorization of $f$. Call Algorithm 4.2 on each square-free factor and
    return the result.
```

Algorithm 4.2: Main Factorization Algorithm

```
Require: Square-free $f \in L[x_1, x_2, \ldots, x_v]$ where $\mathrm{cont}_{x_1}(f) = 1$ and $f \notin L$.
Ensure: Factorization of $f$: $f = l \times f_1 \times f_2 \times \cdots \times f_n$ where $f_i$ is monic and $l = \mathrm{lc}_{x_1,\ldots,x_v}(f)$.
1: Compute $\tilde{f}$ (See Definition 4.3).
2: Compute $\mathbb{D}$, an integral multiple of the defect of the algebraic field $L$ {See Theorem 4.9}.
```

```
3: if $v = 1$ {univariate case} then
4:     Call algorithm 4.4 on $\tilde{f}$ and $\mathbb{D}$ and return the result.
5: end if
6: Choose a heuristic bound $B$ for the largest integer coefficient in the factors of $\tilde{f}$.
7: Let $T = \max_{i=1}^{k} (\deg_{t_i}(\tilde{f}) + \sum_{j=1}^{r} \deg_{t_i} \tilde{m}_j)$
   {Heuristic bound on the degree of $\tilde{f}$ in any parameter: Abbott's conjecture 4.24}
8: Factor $\mathrm{lc}_{x_1}(\tilde{f}) \in L[x_2, \ldots, x_v]$ by calling algorithm efactor. Let $\mathrm{lc}_{x_1}(\tilde{f}) = \gamma \times l_1^{e_1} \times l_2^{e_2} \times \cdots \times l_m^{e_m}$ where $l_i$ is monic.
9: Compute $\tilde{l}_i$. Find $\bar{\gamma}, \bar{D} \in \mathbb{Z}[t_1, \ldots, t_k]$ s.t. $\bar{D} \times \mathrm{lc}_{x_1}(\tilde{f}) = \bar{\gamma} \times \prod_{i=1}^{m} \mathrm{lc}_{x_2,\ldots,x_v}(\tilde{l}_i)$. Update $\tilde{f} := \bar{D} \times \tilde{f}$. {Note $\bar{D} \mid \mathbb{D}^c$ for some $c \in \mathbb{Z}^+$}.
10: Let $N$ be a counter initialized with zero.
11: Main Loop: Choose a new good evaluation point $\alpha = (t_1 = \alpha_1, t_2 = \alpha_2, \ldots, t_k = \alpha_k, x_2 = \beta_2, \ldots, x_v = \beta_v)$ at random that satisfies the requirements of Definition 4.14 in Section 4.2.2. {$\alpha_1, \ldots, \alpha_k, \beta_2, \ldots, \beta_k \in \mathbb{Z} \backslash \{0\}$.}
12: Using Trager's algorithm factor $\tilde{f}(\alpha)$ over $L(\alpha)$ to obtain $\tilde{f}(\alpha) = \Omega' \times u_1 \times \cdots \times u_n$ where $\Omega' = \mathrm{lc}_{x_1}(\tilde{f})(\alpha) \in \mathbb{Q}[z_1, \ldots, z_r]$. If $n = 1$ then return $l \times \mathrm{monic}(\tilde{f})$ {$\tilde{f}$ is irreducible}
13: Using Algorithm 4.6 on inputs $\{u_1, \ldots, u_n\}$, $\mathrm{lc}_{x_1}(\tilde{f}) = \bar{\gamma} \times \tilde{l}_1^{e_1} \times \tilde{l}_2^{e_2} \times \cdots \times \tilde{l}_m^{e_m}$, the evaluation point $\alpha$, $\mathbb{D}$ and $\{D_1, \ldots, D_m\}$ where $D_i = \mathrm{den}(\tilde{l}_i(\alpha)^{-1})$, compute the true leading coefficients of each univariate factor $\{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_n\}$. If this fails set $N := N + 1$. Note that $\tilde{f}$ may be updated in order to distribute the integer content of $\mathbb{D} \times \mathrm{lc}_{x_1}(\tilde{f})$. If $N < 10$ then go to step 11 otherwise shift the variables $x_2, \ldots, x_v$ in $\tilde{f}$, call Algorithm 4.2 recursively on the shifted $\tilde{f}$, and undo the shift in the factors and return. {See Example 4.17}.
14: Compute $\delta, \hat{l} \in \mathbb{Z}[t_1, \ldots, t_k]$ s.t. $\delta \times \mathrm{lc}_{x_1,\ldots,x_v}(\tilde{f}) = \hat{l} \times \prod_{i=1}^{n} \mathrm{lc}_{x_2,\ldots,x_v}(\bar{l}_i)$. {$\delta \mid \mathbb{D}^c$ for some $c \in \mathbb{Z}^+$ and $\hat{l}$ is a factor of $\mathrm{lc}_{x_1,\ldots,x_v}(\tilde{f})$ that is not in $l_1, \ldots, l_n$}.
15: Set $\tilde{f} := \delta \tilde{f}$. At this point we have
        $$\tilde{f}(\alpha) = \hat{l}(\alpha) \times (\bar{l}_1(\alpha) u_1) \times \cdots \times (\bar{l}_n(\alpha) u_n).$$
16: Choose a new good prime $p$ satisfying $\mathrm{lc}_{x_1}(\tilde{f}(\alpha)) \bmod p \neq 0$, $\mathrm{lc}_{z_i}(\tilde{m}_i(\alpha)) \bmod p \neq 0$ and $\tilde{f}(\alpha)$ is square-free modulo $p$.
17: Now we are set for Hensel lifting. Using Hensel lifting on inputs $\tilde{f}, \hat{l}$, the set of univariate images $\{u_1, \ldots, u_n\}$, the set of corresponding true leading coefficients $\{\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_n\}$, the prime $p$, the bound $T$ and the evaluation point $\alpha$, lift the variables $x_2, x_3, \ldots, x_v$ and
```

```
    the parameters $t_1, \dots, t_k$ to obtain $\tilde{f} = \hat{l} \times \bar{f}_1 \times \bar{f}_2 \times \cdots \times \bar{f}_n \bmod \langle \tilde{m}_1, \dots, \tilde{m}_r, p\rangle$ and
    $\bar{l}_i = \mathrm{lc}_{x_1}(\tilde{f}_i)$.
18: If Hensel lifting fails then Set $T := 2 \times T$ and go to Step 11.
19: Call algorithm 4.3 on inputs $\tilde{f}, \bar{f}_1, \bar{f}_2, \dots, \bar{f}_n, \hat{l}$, the prime $p$, the bound $B$ and the set
    $\{l_1, l_2, \dots, l_n\}$. If this fails, set $B := B^2$ and go to step 11 otherwise let $f'_1, f'_2, \dots, f'_n$ be
    the output s.t. $\tilde{f} = \hat{l} \times f'_1 \times \cdots \times f'_n$ over $L$.
20: **return** $\mathrm{lc}_{x_1,\dots,x_v}(f) \times \mathrm{monic}(f'_1) \times \cdots \times \mathrm{monic}(f'_n)$
```

Algorithm 4.3: Sparse p-adic lifting

```
**Require:** $\tilde{f}, \tilde{f}_1, \dots, \tilde{f}_n \in L[x_1, \dots, x_v]$, $\hat{l} \in \mathbb{Z}[t_1, \dots, t_k]$ and $p$ s.t. $\tilde{f} - \hat{l} \times \tilde{f}_1 \times \tilde{f}_2 \times \cdots \times \tilde{f}_n =$
    $0 \bmod \langle \tilde{m}_1, \dots, \tilde{m}_r, p\rangle$. The numerical bound $B$ and $\{l_1, \dots, l_n\}$ the set of the leading
    coefficients of the factors.
**Ensure:** Either FAIL, if the evaluation point is unlucky or polynomials $h_1, h_2, \dots, h_n$ s.t.
    $\tilde{f} = \hat{l} \times h_1 \times \cdots \times h_n$ over $L$.
Let $h_i$ be $\tilde{f}_i$ with its leading coefficient replaced by $l_i$.
Let $e = \tilde{f} - \hat{l} \times h_1 \times \cdots \times h_n \bmod \langle \tilde{m}_1, \dots, \tilde{m}_r\rangle$. {Note that $\deg_{x_1}(e) < \deg_{x_1}(\tilde{f})$ because
    we imposed $\mathrm{lc}_{x_1}(\tilde{f}) = \hat{l} \times \prod_{i=1} nh_i$}
Let $P = p$.
Suppose $\tilde{f}_i = \sum_{j=1}^{T_i} a_{ij} M_{ij}$ with $a_{ij} \in \mathbb{Z}_p$ and $M_{ij}$ monomials.
Let $\sigma_i = \sum_{j=1}^{T_i} A_{ij} M_{ij}$ where $A_{ij}$ is an unknown coefficient.
**while** $e \neq 0$ and $P < 2B$ **do**
    $e' = e/P$ {exact integer division}
    Let $p_z = e' - \hat{l} \times \sum_{i=1}^{n} \sigma_i \frac{\prod_{j=1}^{n} h_j}{h_i} \bmod \langle \tilde{m}_1, \dots, \tilde{m}_r\rangle$.
    Solve for $A_{ij}$'s by collecting and equating coefficients of $p_z$ in $x_1, \dots, x_v$, $t_1, \dots, t_k$ and
        $z_1, \dots, z_r$ to zero modulo $P$.
    If the system of linear equations is inconsistent then return FAIL. {Annihilated term
        in the form due to the unlucky choice of $p$}
    Update $h_i := h_i + \sigma_i \times P$ for $1 \le i \le n$.
    Set $P := P^2$
    Set $e = \tilde{f} - \hat{l} \times h_1 \times \cdots \times h_n \bmod \langle \tilde{m}_1, \dots, \tilde{m}_r\rangle$.
**end while**
If $e = 0$ then return $h_1, h_2, \dots, h_n$ else return FAIL.
```

Algorithm 4.4: Univariate Factorization

```
Require: Square-free $f \in L[x_1]$ and $\mathbb{D}$ the defect of $L$.
Ensure: Unique factorization of $f = \mathrm{lc}_{x_1}(f) \times f_1 \times f_2 \times \cdots \times f_n$ over $L$ s.t. $f_i$ is monic in $x_1$.
1: Compute $\tilde{f}$ (See Definition 4.3) and Let $\bar{l} = \mathrm{lc}_{x_1}(\tilde{f})$.
2: Choose a heuristic bound $B$ on the integer coefficients of the factors of $\tilde{f}$.
3: Let $T = \max_{i=1}^{k}(\deg_{t_i}(\tilde{f}) + \sum_{j=1}^{r} \deg_{t_i} \tilde{m}_j)$
   {Heuristic bound on the degree of $\tilde{f}$ in any parameter: Abbott's conjecture}.
4: Factor $\gamma = \mathbb{D} \times \bar{l} \in \mathbb{Z}[t_1, \ldots, t_k]$ over $\mathbb{Z}$ to obtain $\gamma = \Omega \times \beta_1^{c_1} \times \cdots \times \beta_{k'}^{c_{k'}}$.
5: Main Loop: Choose a new good evaluation point $\alpha = (t_1 = \alpha_1, \ldots, t_k = \alpha_k)$ that satisfies the requirements of definition 4.14.
6: Using Trager's algorithm, factor $h = \tilde{f}(\alpha) = \bar{l}(\alpha) \times h_1 \times h_2 \times \cdots \times h_n$ over the algebraic number field. Note that $\mathrm{lc}_{x_1}(h_i) = 1$.
7: Compute $\tilde{h}_i$ and let $\bar{d}_i = \mathrm{lc}_{x_1}(h_i) \in \mathbb{Z}$. Find the biggest $e_{ij}$ s.t. $\beta_i^{e_{ij}} \mid \bar{d}_j$. Let $l_i = \beta_1^{e_{1i}} \times \cdots \times \beta_{k'}^{e_{k'i}}$. Distribute $\Omega \in \mathbb{Z}$ to $l_i$'s and if needed, update $\tilde{f}$ and $\tilde{h}_i$. At this point we have $l_i = \mathrm{lc}_{x_1}(\tilde{f}_i)$.
8: Compute $\delta, \hat{l} \in \mathbb{Z}[t_1, \ldots, t_k]$ s.t. $\delta \times \bar{l} = \hat{l} \times \prod_{i=1}^{n} l_i$. {$\delta \mid \mathbb{D}^c$ for some $c \in \mathbb{Z}$ and $\hat{l}$ is a factor of $\mathrm{lc}_{x_1}(\tilde{f})$ that is not in $l_1, \ldots, l_n$}
9: Let $\hat{f} = \delta \tilde{f}$ {$\hat{f}(\alpha) = \hat{l}(\alpha) \times \tilde{h}_1 \times \tilde{h}_2 \times \cdots \times \tilde{h}_n$}.
10: Choose a new good prime $p$ satisfying $\mathrm{lc}_{x_1}(\tilde{f}(\alpha)) \bmod p \neq 0$, $\mathrm{lc}_{z_i}(\tilde{m}_i(\alpha)) \bmod p \neq 0$ and $\tilde{f}(\alpha)$ is square-free modulo $p$.
11: Lift the parameters $\{t_1, \ldots, t_k\}$ in $\hat{f}(\alpha) - \hat{l} \times \tilde{h}_1 \times \tilde{h}_2 \times \cdots \times \tilde{h}_n \equiv 0 \bmod p$ using Hensel lifting with $l_i \in \mathbb{Z}[t_1, \ldots, t_k]$ as the true leading coefficient of $\tilde{h}_i$ and $T$ as the degree bound. If this fails, set $T := 2 \times T$ and go to step 5 {unlucky evaluation point}.
12: Call algorithm 4.3 on inputs $\hat{f}, \tilde{h}_1, \tilde{h}_2, \ldots, \tilde{h}_n, \hat{l}$, the prime $p$, $\{l_1, \ldots, l_n\}$ and $B$. If this fails, set $B := B^2$ and go to step 5 (main loop) otherwise let $f'_1, f'_2, \ldots, f'_n$ be the output s.t. $\hat{f} = \hat{l} \times f'_1 \times \cdots \times f'_n$ over $L$.
13: return $\mathrm{lc}_{x_1}(f) \times \mathrm{monic}(f'_1) \times \cdots \times \mathrm{monic}(f'_n)$.
```

Algorithm 4.5: Distinct prime divisors (Similar to Wang [77])

```
Require: A set $\{a_1, a_2, \ldots, a_n\}$ where $a_i \in \mathbb{Z}$.
```

```
Ensure: Either FAIL or a set of divisors $\{d_1, d_2, \ldots, d_n\} \in \mathbb{N}^n$ s.t. $d_i \neq 1$ and $d_i \mid a_i$ and
    $\forall j \neq i : \gcd(d_i, d_j) = 1$.
for $i$ from 1 to $n$ do
    Let $d_i = a_i$.
    for $j$ from 1 to $i - 1$ do
        Let $g = \gcd(d_i, d_j)$.
        Set $d_i := d_i/g$ and $d_j := d_j/g$.
        Let $g_1 = \gcd(g, d_i)$ and $g_2 = \gcd(g, d_j)$. {Either $g_1 = 1$ or $g_2 = 1$}
        while $g_1 \neq 1$ do
            Let $g_1 = \gcd(d_i, g_1)$.
            Set $d_i := d_i/g_1$.
        end while
        while $g_2 \neq 1$ do
            Let $g_2 = \gcd(d_j, g_2)$.
            Set $d_j := d_j/g_2$.
        end while
        if $d_i = 1$ or $d_j = 1$ then
            return FAIL.
        end if
    end for
end for
return $\{d_1, \ldots, d_n\}$.
```

Algorithm 4.6: Distributing leading coefficients

```
Require: $\tilde{f}$ and $U = \{u_1, u_2, \ldots, u_n\}$, the set of monic univariate factors where $u_i \in$
    $L(\alpha)[x_1]$. $l = \gamma \times l_1^{e_1} \times l_2^{e_2} \times \cdots \times l_m^{e_m}$ the *non-monic* factorization of $l = \mathrm{lc}_{x_1}(\tilde{f})$
    where $\gamma \in \mathbb{Z}[t_1, \ldots, t_k]$. The evaluation point $\alpha$ and $\mathbb{D}$ the defect of the algebraic field.
    $\{D_1, \ldots, D_m\}$ where $D_i = \mathrm{den}(l_i(\alpha)^{-1})$.
Ensure: Either FAIL, if the leading coefficient is unlucky or $\{\hat{l}_1, \hat{l}_2, \ldots, \hat{l}_n\}$ where $\hat{l}_i \in$
    $L[x_2, \ldots, x_v]$ is the true leading coefficient of $u_i$ in $x_1$ together with possibly updated $\tilde{f}$.
Let $\beta = \mathbb{D} \times \gamma = \Omega \times \beta_1^{c_1} \times \beta_2^{c_2} \times \cdots \times \beta_{k'}^{c_{k'}}$ where $\Omega \in \mathbb{Z}$.
Let $d_i = \mathrm{den}(u_i)$ and $\mu_i = \beta_i(\alpha)$.
```

```
Apply algorithm 4.5 on input $[D_1, \dots, D_m, \mu_1, \dots, \mu_{k'}]$. If Algorithm 4.5 fails, return
  FAIL. Otherwise let $\{p_1, \dots, p_m, q_1, \dots, q_{k'}\}$ be the output.
For all $1 \le i \le m$, let $g_i = \gcd(\Omega, p_i)$ and Set $p_i := p_i/g_i$. If $p_i = 1$ then return FAIL.
For all $1 \le i \le k'$, let $g'_i = \gcd(\Omega, q_i)$ and Set $q_i := q_i/g'_i$. If $q_i = 1$ then return FAIL.
for each $d_j$ do
  for $i$ from 1 to $m$ do
    Let $g_1 = \gcd(d_j, p_i)$.
    Set $e'_{ji} = 0$.
    while $g_1 \neq 1$ do
      Set $e'_{ji} := e'_{ji} + 1$.
      Set $d_j = d_j/g_1$.
      Set $g_1 = \gcd(d_j, g_1)$.
    end while
  end for
  for $i$ from 1 to $k'$ do
    Let $g_2 = \gcd(d_j, q_i)$.
    Set $c'_{ji} = 0$.
    while $g_2 \neq 1$ do
      Set $c'_{ji} := c'_{ji} + 1$.
      Set $d_j = d_j/g_2$.
      Set $g_2 = \gcd(d_j, g_2)$.
    end while
  end for
end for
for $i$ from 1 to $m$ do
  if $\sum_{j=1}^{n} e'_{ji} \neq e_i$ then return FAIL.
end for
Let $\hat{l}_i = \beta_1^{c_{i1}} \beta_2^{c_{i2}} \dots \beta_{k'}^{c_{ik'}} l_1^{e_{i1}} l_2^{e_{i2}} \dots l_m^{e_{im}}$. Distribute $\Omega \in \mathbb{Z}$ to $\hat{l}_i$s and if needed update $\tilde{f}$.
return $\{\hat{l}_1, \hat{l}_2, \dots, \hat{l}_n\}$.
```

Remark 4.26. In our implementation of algorithm4.2, we first choose an evaluation point and compute a univariate factorization then factor $\mathrm{lc}_{x_1}(\tilde{f})$. This is because if $\tilde{f}$ is irreducible, then we do not bother factoring the leading coefficient which might be a big polynomial.

Description of Algorithm 4.3

The input to algorithm 4.3 is

$$\tilde{f} - \hat{l} \times \tilde{f}_1 \times \tilde{f}_2 \times \cdots \times \tilde{f}_n \equiv 0 \bmod \langle p, \tilde{m}_1, \ldots, \tilde{m}_r \rangle .$$

Note, if $\tilde{m}_i$ is not monic, the reduction modulo $\{\tilde{m}_1, \ldots, \tilde{m}_r\}$ does not introduce fractions in the parameters because of $\hat{l}$. Let $e_1 = \tilde{f} - \hat{l} \times \tilde{f}_1 \times \cdots \times \tilde{f}_n \bmod \langle \tilde{m}_1, \ldots, \tilde{m}_r \rangle$. We know that $p \mid e_1$. If $e_1 = 0$ then we are done. We want to find polynomials $\sigma_1, \ldots, \sigma_n$ s.t.

$$\tilde{f} - \hat{l} \times (\tilde{f}_1 + \sigma_1 p)(\tilde{f}_2 + \sigma_2 p) \ldots (\tilde{f}_n + \sigma_n p) \equiv 0 \bmod p^2.$$

Expanding the above expression modulo $\langle p, \tilde{m}_1, \ldots, \tilde{m}_r \rangle$ results in $g \equiv 0 \bmod p$ where

$$g = \hat{l} \times (\sigma_1 \tilde{f}_2 \tilde{f}_3 \ldots \tilde{f}_n + \cdots + \sigma_n \tilde{f}_1 \tilde{f}_2 \ldots \tilde{f}_{n-1}) - \frac{e}{p}.$$

We assume that the monomials in σ_i are the same as the monomials in $\tilde{f}_i$ with the integer coefficient replaced by an unknown. We compute the polynomial g and equate each coefficient in $z_1, \ldots, z_r, t_1, \ldots, t_k, x_1, \ldots, x_v$ to zero. This gives us a linear system which has a unique solution because we already know the exact leading coefficient in the main variable of each factor $\tilde{f}_i$ and uniqueness is guaranteed by Hensel's lemma. After solving this system we will obtain the correction polynomials $\sigma_1, \ldots, \sigma_n$. We update each factor $\tilde{f}_i := \tilde{f}_i + \sigma_i p$. Now we have

$$\tilde{f} - \hat{l} \times \tilde{f}_1 \times \tilde{f}_2 \times \cdots \times \tilde{f}_n \equiv 0 \bmod p^2.$$

We repeat this non-linear lifting algorithm until $p^{2^k} > 2|B|$ where B is the heuristic bound chosen in Algorithm 4.2 for the integer coefficients in the factors of $\tilde{f}$. Thus if there are no extraneous factors and no annihilated terms caused by the choice of primes and evaluation points, the algorithm will not depend on a bound on the size of the coefficients in the factor of $\tilde{f}$ which could be big.

Remark 4.27. In Step 8 of Algorithm 4.4 and Step 14 of algorithm 4.2, we compute $\hat{l} \in \mathbb{Z}[t_1, \ldots, t_k]$ which is the factor of the leading coefficient of $\tilde{f}$ in all the variables which does not show up in the leading coefficient of any factors of $\tilde{f}$.

Remark 4.28. The bottleneck of Hensel lifting algorithm is solving the Diophantine equations. One can solve these Diophantine equations using sparse interpolation with a similar technique as in algorithm 4.3. Here is an example.

Example 4.29. Let $\breve{m} = z^2 - (t-1)^3$ and

$$\tilde{f} = \left(t^3 - t - t^2 + 1\right) x^2 - x\,(2\,t+1)\,z - t^4 + t^2.$$

Suppose we choose the evaluation point to be $t = 4$. We compute the univariate factors using Trager's algorithm and after computing and attaching the leading coefficients of the factors we have

$$\begin{aligned}
\hat{f} &= (t-1)^2 \tilde{f},\\
\tilde{f}_1 &= \left(t^3 - t - t^2 + 1\right) x + 16\,z,\\
\tilde{f}_2 &= \left(t^2 - 2\,t + 1\right) x - 5\,z,
\end{aligned}$$

where $\hat{f} - \tilde{f}_1\tilde{f}_2 \equiv 0 \bmod (t-4)$. Now we start Hensel lifting. The first error polynomial is $e_1 = \hat{f} - \tilde{f}_1\tilde{f}_2$. We have

$$\frac{e_1}{t-4} = \left(3\,t^2 z - 6\,tz + 3\,z\right) x - t^5 - 2\,t^4 - 8\,t^3 + 46\,t^2 - 55\,t + 20.$$

Now we need to find two polynomials σ_1 and σ_2 s.t.

$$\sigma_2\tilde{f}_1 + \sigma_1\tilde{f}_2 - \frac{e_1}{t-4} \equiv 0 \bmod (t-4). \tag{4.1}$$

Similar to algorithm 4.3, we can assume that σ_1 and σ_2 have the same monomials as $\tilde{f}_1$ and $\tilde{f}_2$ respectively and since we know that the leading coefficient of $\tilde{f}_1$ and $\tilde{f}_2$ are correct, the forms for σ_1 and σ_2 are $\sigma_1 = Az$ and $\sigma_2 = Bz$. Using these forms and Equation 4.1 we construct and solve a linear system to obtain $A = 8, B = -1$. We update $\tilde{f}_1 := \tilde{f}_1 + \sigma_1 \times (t-4)$ and $\tilde{f}_2 := \tilde{f}_2 + \sigma_2 \times (t-4)$ to get

$$\begin{aligned}
\tilde{f}_1 &= (t^3 - t^2 - t + 1)x + 16z + 8(t-4)z,\\
\tilde{f}_2 &= (t^2 - 2t + 1)x - (t-4)z - 5z.
\end{aligned}$$

This time the new error polynomial is $e_2 = \hat{f} - \tilde{f}_1\tilde{f}_2$ and we have

$$\frac{e_2}{(t-4)^2} = \left(t^2 z - 2\,tz + z\right) x - t^4 + 2\,t^3 - 2\,t + 1$$

and

$$\sigma_2\tilde{f}_1 + \sigma_1\tilde{f}_2 - \frac{e_2}{(t-4)^2} \equiv 0 \bmod (t-4)^2. \tag{4.2}$$

The new assumed forms are

$$\sigma_1 = Az + Bz(t-4),$$
$$\sigma_2 = Czt + Dz.$$

Again we construct a system of linear equations using Equation 4.2 and after solving this system we have $A = 1, B = 0, C = 0, D = 0$. We update $\tilde{f}_1$ and $\tilde{f}_2$ and to obtain

$$\tilde{f}_1 = \left(t^3 - t^2 - t + 1\right)x + zt^2,$$
$$\tilde{f}_2 = \left(t^2 - 2\,t + 1\right)x - zt - z.$$

The new error polynomial $e_3 = \hat{f} - \tilde{f}_1\tilde{f}_2$ is zero so $\tilde{f} = \mathrm{lc}_x(\tilde{f}) \times \mathrm{monic}(\tilde{f}_1) \times \mathrm{monic}(\tilde{f}_2)$ and we are done.

We do not use this method in our new algorithm for lifting parameters and variables. This is because it was always slower than solving the Diophantine equations using the traditional method. The reasons are:

1. The systems of linear equations in each step can be very big if the factors are dense.

   Example 4.30. Suppose $\tilde{f}_1$, $\tilde{f}_2$ and $\tilde{f}_1 \times \tilde{f}_2$ have N_1, N_2 and N terms respectively. Then the system of linear equation has N equations and as many as $N_1 - 1 + N_2 - 1$ unknowns.

2. As Hensel lifting progresses, usually, the error term gets smaller so solving the Diophantine equation is usually easier at the next step. But using sparse interpolation, as the Hensel lifting algorithm proceeds, each factor $\tilde{f}_i$ usually gets bigger because we add new terms, so the system of linear equations gets bigger which makes the Hensel lifting slower.

We do not have the second problem above for sparse interpolation in algorithm 4.3, when we lift integer coefficients, mainly because the forms of the σ polynomials do not change due to the fact that only integer coefficients of factors of $\tilde{f}$ are being updated.

4.4 Benchmarks

We have compared Maple 13 implementations of our new algorithm (efactor) with Maple's implementation of Trager's algorithm modified to use SparseModGcd (See [31]) for computing GCDs over L. This modified Maple implementation of Trager's algorithm is more efficient (See [56]). We should also mention that the Magma computer algebra system also uses Trager's algorithm for factorization over algebraic fields.

The eight benchmark problems are given here.

1. Problem 1:

$$\begin{gathered}
f_1 = 63x^2yt + 16x^2t^2 + 7xz_1{}^3 - 43y^2t^2 - 34yz_1{}^2z_2 - \\
20xyz_1{}^2z_2 - 35y^2z_2x^2z_1 + 29y^2x^2t^3z_2 - 27y^2x^2tz_1{}^3 + \\
78y^2x^2tz_1z_2{}^3 + 25y^2x^2tz_2{}^4 + 30y^2x^2z_2{}^5, \\
f_2 = -27x - 99yz_1 - 81xy^2t - 42t^3z_2 + 30xyz_1{}^3 - \\
21yz_1{}^4 - 85y^2z_2x + 50y^2xz_1{}^2z_2{}^2 - 55y^2xz_2{}^4 - \\
64y^2xz_1t^2z_2{}^2 - 75y^2xtz_1z_2{}^3 + 90y^2xz_1{}^3z_2{}^2
\end{gathered}$$

2. Problem 2:

$$\begin{gathered}
f_1 = -51z_1{}^2 + 77xz_1z_2{}^2 + 95x^4z_2 + x^3z_1z_2 + \\
55xtz_2{}^3 + 53x^4y^2z_1z_2 - 28x^4y^2z_2{}^2 + 5x^4y^2t^2z_1 + \\
xy^2tz_2 + 13x^4y^2tz_1z_2 - 10x^4y^2t^2z_1{}^2z_2 - 82x^4y^2z_1{}^4z_2, \\
f_2 = -15xy^3 - 59xy^2t - 96t^2z_1z_2 + 72x^4z_1 - 87ytz_2{}^3 + \\
98x^4y^3z_1{}^3 - 48x^4y^3t^3z_1z_2 - 19x^4y^3t^2z_1{}^2z_2 + \\
47t^2z_1{}^2z_2 + 62x^4y^3t^2z_2{}^3 + 37x^4y^3z_1{}^4z_2 + 5x^4y^3z_1z_2{}^4
\end{gathered}$$

3. Problem 3:

$$\begin{aligned}
f_1 &= x^2 + \left(3 + z_2z_1y + t^3\right)x + y^2 + z_2{}^2s, \\
f_2 &= 2tx^2 + \left(-z_2s^3y + z_2{}^2s\right)x + 5z_1t^3 - 3sy^2.
\end{aligned}$$

4. Problem 4:

$$\begin{aligned}
f_1 &= x^3 + y^3 + z_2z_1xy^2 + t^3x^2 + x + z_2{}^2s, \\
f_2 &= 2tx^3 - 3sy^3 - z_2s^2txy^2 + z_2{}^2sx + 5z_1t^3.
\end{aligned}$$

5. Problem 5:

$$\begin{aligned}
f_1 &= x^2 + (t^3 + 3z_2z_1y + wt)x + 3swz_2 + y^2 + w^2 + {z_2}^2s, \\
f_2 &= 2tx^2 + ({z_2}^2s - z_2s^3y)x + 5z_1t^3 - 3sy^2 - 2wyz_2z_1 + stw^2.
\end{aligned}$$

6. Problem 6:

$$\begin{aligned}
f_1 = &\frac{19}{2}c_4^2 - \sqrt{11}\sqrt{5}\sqrt{2}c_5c_4 - 2\sqrt{5}c_1c_2 - 6\sqrt{2}c_3c_4 + \frac{3}{2}c_0^2 + \frac{23}{2}c_5^2 \\
&+\frac{7}{2}c_1^2 - \sqrt{7}\sqrt{3}\sqrt{2}c_3c_2 + \frac{11}{2}c_2^2 - \sqrt{3}\sqrt{2}c_0c_1 + \frac{15}{2}c_3^2 - \frac{10681741}{1985}, \\
f_2 = &\frac{19}{2}c_4^2 - \sqrt{11}\sqrt{5}\sqrt{2}c_5c_4 - 2\sqrt{5}c_1c_2 + 6\sqrt{2}c_3c_4 + \frac{3}{2}c_0^2 + \frac{23}{2}c_5^2 \\
&+\frac{7}{2}c_1^2 + \sqrt{7}\sqrt{3}\sqrt{2}c_3c_2 + \frac{11}{2}c_2^2 + \sqrt{3}\sqrt{2}c_0c_1 + \frac{15}{2}c_3^2 - \frac{10681741}{1985}
\end{aligned}$$

7. Problem 7:

```
>f1 := randpoly([x,y,u,v,w,t,z1,z2], terms = 200);
>f2 := randpoly([x,y,u,v,w,t,z1,z2], terms = 200);
```

8. Problem 8:

```
>f1 := s*x^50 + randpoly([x,z,t,s], degree = 50, coeffs = rand(10^50)) + 1;
>f2 := t*x^50 + randpoly([x,z,t,s], degree = 50, coeffs = rand(10^50)) + 1;
```

The minimal polynomials are

1. Problems 1,2 and 7:

$$\begin{aligned}
m_1 &= {z_1}^2 - t, \\
m_2 &= {z_2}^3 - z_1{z_2}^2 - t^2z_2 + 7.
\end{aligned}$$

2. Problems 3 to 5:

$$\begin{aligned}
m_1 &= {z_1}^2 - 2, \\
m_2 &= {z_2}^3 + t{z_2}^2 + s.
\end{aligned}$$

3. Problem 8:

$$m_1 = z^2 - tz - s.$$

The timings are given in Table 4.1. All timings are in CPU seconds and were obtained on Maple 13 on a 64 bit AMD Opteron CPU @ 2.4 GHz, running Linux. In the table, n is the number of variables, r is the number of field extensions, k is the number of parameters, d is the total degree of f, $\#f$ is the number of terms in f and $\#\tilde{f}$ is the number of terms in $\tilde{f}$. In all the problems, f factors into two irreducible factors f_1 and f_2.

Problems 1 and 2 have large leading coefficients in the main variable x. Problems 3–5 illustrate how Trager's algorithm is sensitive to the degree of the input and the number of variables. Problem 7 has many variables and parameters. Problem 8 has large integer coefficients. For problem 6, we multiplied the polynomial f from Example 1.33 by one of its conjugates. Table 4.1 illustrates that Trager's algorithm did not finish in 50,000 seconds. In fact Maple had not computed the norm of the input polynomial after $50,000$ seconds.

For each problem we used $p = 3037000453$, a 31.5 bit prime, for Hensel lifting. For problems 3,4,5 and 7, p is big enough so that there is no need to lift the integer coefficients using sparse p-adic lifting algorithm. For problems 1,2 and 6, the number of lifting steps is one, i.e., $p^2 > 2||\tilde{f}_i||_\infty$. For the problem 8, the number of lifting steps is three, i.e. $p^8 > 2||\tilde{f}_i||_\infty$.

The last column in Table 4.1 is the time for computing

$$\gcd(f_1 f_2, f_1(f_2+1))$$

using our SparseModGcd algorithm in [31]. One can see that our factorization algorithm is often as fast as the GCD algorithm on a problem of comparable size, except for problem 6. In problem 6, almost all (99%) of the time was factoring the univariate polynomial over $\mathbb{Q}(\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{7}, \sqrt{11})$ using Trager's algorithm.

The percentages of timings for different parts of our new algorithm for these problems are presented in Table 4.2. In this table, the second column is the percentage of time spent on univariate factorization over $L(\alpha)$ using Trager's algorithm. The numbers in the third column correspond to the time spent on lifting variables and integer coefficients respectively. And finally, numbers in the last column are the percentages of time spent on doing square-free factorizations of the inputs. One can see that the bottleneck of our new algorithm for the first two problems is the sparse p-adic lifting algorithm. This is because of the large number of terms and large integers in $\tilde{f}$.

#	$(n, r, k, d, \#f, \#\tilde{f})$	Trager	efactor	GCD
1	(2,2,1,17,191,6408)	5500	259.91	47.47
2	(2,2,1,22,228,12008)	37800	296.74	56.90
3	(2,2,2,10,34,34)	120	0.22	0.16
4	(2,2,2,12,34,34)	571	0.31	0.19
5	(3,2,2,10,69,69)	5953	0.27	0.29
6	(6,5,0,4,46,46)	> 50000	88.43	1.93
7	(5,2,1,10,15489,17052)	> 50000	58.41	57.75
8	(1,1,2,102,426,928)	16427	72.10	7.71

Table 4.1: Timings (in CPU seconds)

#	Univariate	Lifting	Sqr-free
1	0.30%	(4.99%,90.1%)	4.01%
2	0.80%	(7.82%,84.42%)	6.45%
3	51.61%	(17.05%,0%)	19.35%
4	57.23%	(22.03%,0%)	12.50%
5	42.86%	(35.53%,0%)	19.41%
6	99.47%	(0.31%,0.52%)	0.14%
7	0.80%	(28.9%,0%)	67.41%
8	2.06 %	(91.68%,5.47%)	0.70 %

Table 4.2: Timing (percentile) for different parts of efactor

Chapter 5

Summary and Future Work

Within this dissertation we presented efficient algorithms for computations with sparse polynomials. We have designed a probabilistic parallel algorithm for interpolating sparse polynomials over a finite field efficiently. We assume the target polynomial is represented with a black box. Our algorithm is a generalization of the Ben-Or and Tiwari algorithm for interpolating polynomials with integer coefficients. We compared this new algorithm to Zippel's sparse interpolation algorithm and the racing algorithm by Kaltofen and Lee. Zippel's algorithm is used in several computer algebra systems such as Maple, Mathematica and Magma. We provided benchmarks illustrating the performance of our algorithm for sparse polynomials. In particular, the benchmarks show that for sparse polynomials, the new algorithm does fewer probes compared to Zippel's algorithm and a comparable number to the racing algorithm. Another advantage of our new algorithm is that it does not interpolate each variable sequentially and hence can be easily parallelized. We have implemented the parallel version of our algorithm in Cilk. One disadvantage of our new algorithm is that it performs poorly on dense polynomials compared to Zippel's algorithm and the racing algorithm.

One of the main applications for a sparse interpolation algorithm is computing greatest common divisors. In particular, we are interested in computing GCDs of polynomials over algebraic function fields. We presented three improvements to the SparseModGcd algorithm given in [31]. One of the main bottlenecks of the SparseModGcd algorithm on sparse inputs is computing the univariate GCDs of polynomials over an algebraic number ring modulo a prime p. We presented a library of in-place algorithms for arithmetic in L_p and $L_p[x]$ where L_p is an algebraic number field with one or more extensions. The main idea is to eliminate all the calls to the storage manager by pre-allocating one large piece of working storage, and re-using parts of it in a computation. We have implemented this library in C programming language and compared it to Maple and Magma computer algebra systems on some benchmarks. This library is now integrated in Maple 14. The second improvement is to prove that we can eliminate the trial divisions in positive characteristic in the SparseModGcd algorithm. These trial divisions are the bottleneck for polynomials with a dense GCD. We do this by giving an upper bound on the probability that the maximal quotient rational reconstruction (MQRR) algorithm fails. Finally we showed that a previous solution to the normalization problem given in [9] has an error. We gave a new solution to this problem and proved that this solution works. The new method for solving the normalization problem has two advantages. Firstly, it enables us to choose the evaluation points in a way that the systems of linear equations in Zippel's sparse interpolation algorithm would be transposed Vandermonde systems and hence can be computed in quadratic time and linear space (compared to cubic time and quadratic space for a general linear system). Secondly, the systems of linear equations in Zippel's interpolation method can now be solved in parallel.

Finally we presented an efficient algorithm for factoring multivariate polynomials over algebraic function fields with multiple field extensions and parameters. Our algorithm uses Hensel lifting and is a generalization of Wang's EEZ algorithm which is used for factoring multivariate polynomials with integer coefficients. In our new algorithm, we evaluate all the parameter and variables (except one variable) at random evaluation points. We factor the evaluated univariate polynomial over an algebraic number field using Trager's algorithm and then use Hensel lifting to interpolate the variables and parameters. The challenging problem here is that one needs to know the exact leading coefficient of each univariate factor in all the variables and parameters for Hensel lifting to succeed. We recursively factor the leading coefficient of the input polynomial in the main variable and provide a method for determining the leading coefficient of each factor. To distribute the factors of the leading coefficient we evaluate all variables and parameters at integers and look for unique relatively prime divisors of the denominators of the inverses, which will be an integer. In Hensel lifting, in order to avoid computations with fractions, one usually does the computations modulo a power of a prime p^l such that p^l is greater than twice the absolute value of the biggest coefficient in the input polynomial and all the factors. But the known bounds on the size of the coefficients in the factors are poor, hence in our new algorithm we do the computations modulo a suitable prime p and then lift the integer coefficients in the factors using a new algorithm called sparse p-adic lifting. This algorithm uses sparse interpolation. We provided benchmarks comparing our algorithm to Trager's algorithm which is used in Maple and Magma. The benchmarks clearly show the superiority of our algorithm. This algorithm is currently being integrated in the next version of Maple (version 16).

Future Work

One future extension point is to use our new sparse interpolation algorithm in the SparseModGcd algorithm. This way the sparse interpolation can be done in parallel. Unlike Zippel's sparse interpolation algorithm, our new interpolation algorithm is not variable by variable, so if used in the GCD algorithm, we need to be able to do multivariate rational function reconstruction efficiently (See e.g. [42]). Another solution is to multiply the monic images of the GCD by an image of $\Delta = \gcd(\mathrm{lc}_{x_1}(f_1), \mathrm{lc}_{x_1}(f_2)) \in L[x_2, \ldots, x_n]$. This way, the interpolated polynomial is $\frac{\Delta}{\mathrm{lc}_{x_1}(g)} g$ which could be much bigger than g.

We did not provide any complexity analysis for our new factorization algorithm in Chapter 4. In this chapter, we impose some conditions on *good* evaluation points but we also need to find an estimate on the size of the evaluation points that we need choose, so that the algorithm works correctly without running in to any problems. We need to prove that provided the algorithm that we use for univariate factorization over an algebraic ring is polynomial-time, then the time complexity of algorithm is polynomial in the size of the input. Efficient polynomial time algorithms for univariate factorization exist (See e.g. [47, 52, 72, 24]). We are also working on a theorem to prove that if the norms of the irreducible factors of the leading coefficient in the main variable do not divide each other, then the factorization algorithm given will terminate and output the irreducible factorization of the input polynomial.

Bibliography

[1] J. A. Abbott. *On the factorization of polynomials over algebraic fields.* PhD thesis, School of Math. Sci., Univ. of Bath, England, 1989.

[2] Holger Bast, Kurt Mehlhorn, Guido Schafer, and Hisao Tamaki. Matching algorithms are fast in sparse random graphs. *Theory of Computing Systems*, 39:3–14, 2006.

[3] Michael Ben-Or and Prasoon Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 301–309, New York, NY, USA, 1988. ACM.

[4] Wieb Bosma, John J. Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3–4):235–266, 1997.

[5] Y.Z. Boutros, G.P. Fiani, and E.S. Looka. Polynomial factorization in $GF(2^m)$. *Engineering Fracture Mechanics*, 47(3):451 – 455, 1994.

[6] R. Bradford. Some results on the defect. In *Proceedings of ISSAC '89*, ISSAC '89, pages 129–135, New York, NY, USA, 1989. ACM.

[7] W. S. Brown. On Euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM*, 18(4):478–504, 1971.

[8] Bruce W. Char, Keith O. Geddes, and Gaston H. Gonnet. GCDHEU: Heuristic polynomial GCD algorithm based on integer GCD computation. *J. Symb. Comput.*, 7(1):31–48, 1989.

[9] Jennifer de Kleine, Michael Monagan, and Allan Wittkopf. Algorithms for the non-monic case of the sparse modular GCD algorithm. In *Proceedings of ISSAC '05*, pages 124–131, New York, NY, USA, 2005. ACM Press.

[10] Angel Díaz and Erich Kaltofen. Foxbox: a system for manipulating symbolic objects in black box representation. In *Proceedings of ISSAC '98*, pages 30–37, New York, NY, USA, 1998. ACM.

[11] Mark J. Encarnación. On a modular algorithm for computing GCDs of polynomials over algebraic number fields. In *Proceedings of ISSAC '94*, pages 58–65. ACM Press: New York, NY, 1994.

[12] Mark J. Encarnación. Computing gcds of polynomials over algebraic number fields. *J. Symb. Comp.*, 20(3):299–313, 1995.

[13] Mark J. Encarnación. Factoring polynomials over algebraic number fields via norms. In *Proceedings of ISSAC '97*, pages 265–270, New York, NY, USA, 1997. ACM.

[14] Richard Fateman. Comparing the speed of programs for sparse polynomial multiplication. *SIGSAM Bull.*, 37(1):4–15, 2003.

[15] T. Finck, G. Heinig, and K. Rost. An inversion formula and fast algorithms for cauchy-vandermonde matrices. *Linear Algebra Appl.*, 183:179–191, 1993.

[16] Shuhong Gao. Factoring multivariate polynomials via partial differential equations. *Math. Comput.*, 72:801–822, April 2003.

[17] Sanchit Garg and Éric Schost. Interpolation of polynomials given by straight-line programs. *Theor. Comput. Sci.*, 410:2659–2662, 2009.

[18] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers: Boston/Dordrecht/London, 2002.

[19] Jürgen Gerhard and Ilias S. Kotsireas. Private communication.

[20] Mark Giesbrecht, Erich Kaltofen, and Wen-shin Lee. Algorithms for computing the sparsest shifts of polynomials via the Berlekamp/Massey algorithm. In *Proceedings of ISSAC '02*, pages 101–108. ACM, 2002.

[21] Mark Giesbrecht and Daniel Roche. Interpolation of shifted-lacunary polynomials. *Computational Complexity*, 19:333–354, 2010.

[22] Dima Yu. Grigoriev, Marek Karpinski, and Michael F. Singer. Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields. *SIAM J. Comput.*, 19(6):1059–1063, 1990.

[23] Dima Yu. Grigoriev and Y. N. Lakshman. Algorithms for computing sparse shifts for multivariate polynomials. In *Proceedings of ISSAC '95*, pages 96–103. ACM, 1995.

[24] W. Hart, M. van Hoeij, and A. Novocin. Practical polynomial factoring in polynomial time. In *Proceedings of ISSAC 2010*, New York, NY, USA, 2010. ACM.

[25] David Harvey and Daniel S. Roche. An in-place truncated Fourier transform and applications to polynomial multiplication. In *Proceedings of ISSAC 2010*, pages 325–329, New York, NY, USA, 2010. ACM.

[26] J. van der Hoeven and G. Lecerf. On the bit-complexity of sparse polynomial and series multiplication. Manuscript, 2010.

[27] Hoon Hong, Andreas Neubacher, and Wolfgang Schreiner. The design of the saclib/paclib kernels. In *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 288–302. Springer Berlin / Heidelberg, 1993.

[28] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[29] Ming-Deh A. Huang and Ashwin J. Rao. Interpolation of sparse multivariate polynomials over large finite fields with applications. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 508–517. Society for Industrial and Applied Mathematics, 1996.

[30] S. M. M. Javadi. Sparse modular GCD algorithm for polynomials over algebraic function fields. Master's thesis, Simon Fraser University, Burnaby, Canada, November 2006.

[31] S. M. Mahdi Javadi and M. B. Monagan. A sparse modular GCD algorithm for polynomials over algebraic function fields. In *Proceedings of ISSAC '07*, pages 187–194, New York, NY, USA, 2007. ACM.

[32] Seyed Mohammad Mahdi Javadi and Michael Monagan. In-place arithmetic for univariate polynomials over an algebraic number field. In *Proceedings of the Joint Conference of ASCM '09 and MACIS '09*, pages 330–341, 2009.

[33] Seyed Mohammad Mahdi Javadi and Michael Monagan. Parallel sparse polynomial interpolation over finite fields. In *Proceedings of PASCO '10*, PASCO '10, pages 160–168, New York, NY, USA, 2010. ACM.

[34] Seyed Mohammad Mahdi Javadi and Michael B. Monagan. On factorization of multivariate polynomials over algebraic number and function fields. In *Proceedings of ISSAC '09*, ISSAC '09, pages 199–206, New York, NY, USA, 2009. ACM.

[35] E. Kaltofen, Y. N. Lakshman, and J.-M. Wiley. Modular rational sparse multivariate polynomial interpolation. In *Proceedings of ISSAC '90*, pages 135–139. ACM, 1990.

[36] E. Kaltofen and B. Trager. Computing with polynomials given by black boxes for their evaluations: greatest common divisors, factorization, separation of numerators and denominators. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:296–305, 1988.

[37] Eric Kaltofen. Private communication.

[38] Erich Kaltofen. Sparse hensel lifting. In *EUROCAL '85: Research Contributions from the European Conference on Computer Algebra-Volume 2*, pages 4–17, London, UK, 1985. Springer-Verlag.

[39] Erich Kaltofen and Yagati N. Lakshman. Improved sparse multivariate polynomial interpolation algorithms. In *Proceedings of ISSAC '88*, pages 467–474. Springer-Verlag, 1989.

[40] Erich Kaltofen and Wen-shin Lee. Early termination in sparse interpolation algorithms. *J. Symb. Comput.*, 36(3-4):365–400, 2003.

[41] Erich Kaltofen, Wen-shin Lee, and Austin A. Lobo. Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. In *Proceedings of ISSAC '00*, pages 192–201. ACM, 2000.

[42] Erich Kaltofen and Zhengfeng Yang. On exact and approximate interpolation of sparse rational functions. In *Proceedings of ISSAC '07*, pages 203–210, New York, NY, USA, 2007. ACM.

[43] Erich L. Kaltofen. Fifteen years after DSC and WLSS2 what parallel computations i do today: invited lecture at pasco 2010. In *Proceedings of PASCO '10*, PASCO '10, pages 10–17, New York, NY, USA, 2010. ACM.

[44] L. Kronecker. Grundzüge einer arithmetischen theorie der algebraischen größen. *J. f. d. reine u. angew. Math.*, 92:1–122, 1882.

[45] Y. N. Lakshman and B. David Saunders. Sparse polynomial interpolation in nonstandard bases. *SIAM J. Comput.*, 24(2):387–397, 1995.

[46] Y. N. Lakshman and B. David Saunders. Sparse shifts for univariate polynomials. *Applicable Algebra in Engineering, Communication and Computing*, 7(5):351–364, 1996.

[47] Susan Landau. Factoring polynomials over algebraic number fields. *SIAM Journal on Computing*, 14(1):184–195, 1985.

[48] Lars Langemyr and Scott McCallum. The computation of polynomial greatest common divisors over an algebraic number field. *J. Symb. Comp.*, 8(5):429–448, 1989.

[49] A. K. Lenstra. Factoring polynomials over algebraic number fields. In *EUROCAL '83*, volume 169, pages 245–254, New York, NY, USA, 1983. Springer-Verlag.

[50] A.K. Lenstra, H.W.jun. Lenstra, and Lászlo Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

[51] Arjen K. Lenstra. Lattices and factorization of polynomials over algebraic number fields. In *EUROCAM '82: Proceedings of the European Computer Algebra Conference on Computer Algebra*, pages 32–39, London, UK, 1982. Springer-Verlag.

[52] Arjen K. Lenstra. Factoring multivariate polynomials over algebraic number fields. *SIAM J. Comput.*, 16(3):591–598, 1987.

[53] Xin Li, Marc Moreno Maza, and Éric Schost. Fast arithmetic for triangular sets: from theory to practice. In *Proceedings of ISSAC '07*, pages 269–276. ACM, 2007.

[54] G. Labahn M. Giesbrecht and W s. Lee. Symbolic-numeric sparse interpolation of multivariate polynomials. *J. Symb. Comput.*, 44:943–959, 2009.

[55] J. L. Massey. Shift-register synthesis and bch decoding. *IEEE Trans. Inf. Theory it-15*, pages 122–127, 1969.

[56] Michael Monagan. Computing polynomial greatest common divisors over algebraic number and function fields. `http://www.cecm.sfu.ca/~pborwein/MITACS/highlights/sparseGcd.pdf`.

[57] Michael Monagan. Maximal quotient rational reconstruction: An almost optimal algorithm for rational reconstruction. In *Proceedings of ISSAC '04*, pages 243–249. ACM Press: New York, NY, 2004.

[58] Michael B. Monagan. In-place arithmetic for polynomials over $\mathbb{Z}_n$. In *DISCO '92: Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 22–34, London, UK, 1993. Springer-Verlag.

[59] Peter Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44(70):519–521, 1985.

[60] Rajeev Motwani. Average-case analysis of algorithms for matchings and related problems. *J. ACM*, 41:1329–1356, November 1994.

[61] *PARI/GP, version 2.3.4*. Bordeaux, 2008. `http://pari.math.u-bordeaux.fr/`.

[62] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Trans. on Info. Theory*, IT-24:106–110, 1978.

[63] Michael O. Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput*, 9:273–280, 1979.

[64] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[65] Allan Steel. Multiplication in $L_p[x]$ in Magma. private communication, 2009.

[66] *Cilk 5.4.6 Reference Manual.* MIT. `http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf`.

[67] Barry M. Trager. Algebraic factoring and rational function integration. In *SYMSAC '76: Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 219–226, New York, NY, USA, 1976. ACM.

[68] B. L. van der Waerden. *Modern Algebra*, volume 1. tr. Fred Blum, Frederick Ungar Publishing Co., New York, 1953.

[69] Mark van Hoeij. Factoring polynomials and the knapsack problem. *Journal of Number Theory*, 95(2):167 – 189, 2002.

[70] Mark van Hoeij and Michael Monagan. A modular gcd algorithm over number fields presented with multiple extensions. In *Proceedings of ISSAC '02*, pages 109–116, New York, NY, USA, 2002. ACM Press.

[71] Mark van Hoeij and Michael Monagan. Algorithms for polynomial gcd computation over algebraic function fields. In *Proceedings of ISSAC '04*, pages 297–304. ACM Press: New York, NY, 2004.

[72] Mark van Hoeij and Andrew Novocin. Gradual sub-lattice reduction and a new complexity for factoring polynomials. In *LATIN 2010: Theoretical Informatics*, volume 6034 of *Lecture Notes in Computer Science*, pages 539–553. Springer Berlin / Heidelberg, 2010.

[73] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra.* Cambridge University Press: Cambridge, New York, Port Melbourne, Madrid, Cape Town, second edition, 2003.

[74] Dongming Wang. A method for factorizing multivariate polynomials over successive algebraic extension fields. Preprint. RISC Linz.

[75] Paul S. Wang. Factoring multivariate polynomials over algebraic number fields in macsyma. *SIGSAM Bull.*, 9(3):21–23, 1975.

[76] Paul S. Wang. Factoring multivariate polynomials over algebraic number fields. *Mathematics of Computation*, 30(134):324–336, 1976.

[77] Paul S. Wang. An improved multivariate polynomial factorization algorithm. *Math. Comp.*, 32(144):1215–1231, 1978.

[78] Paul S. Wang. The EEZ-GCD algorithm. *SIGSAM Bull.*, 14(2):50–60, 1980.

[79] Paul S. Wang. A p-adic algorithm for univariate partial fractions. In *SYMSAC '81: Proceedings of the fourth ACM symposium on Symbolic and algebraic computation*, pages 212–217, New York, NY, USA, 1981. ACM Press.

[80] Paul S. Wang, M. J. T. Guy, and J. H. Davenport. P-adic reconstruction of rational numbers. *SIGSAM Bull.*, 16(2):2–3, 1982.

[81] P. J. Weinberger and L. P. Rothschild. Factoring polynomials over algebraic number fields. *ACM Trans. Math. Softw.*, 2(4):335–350, 1976.

[82] Lihong Zhi. Algebraic factorization and gcd computation. *Mathematics Mechanization and Applications*, pages 325–342, 2000.

[83] R. E. Zippel. *Probabilistic algorithms for sparse polynomials*. PhD thesis, Massachusetts Inst. of Technology, Cambridge, USA, September 1979.

[84] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *EUROSAM '79: Proceedings of the International Symposiumon on Symbolic and Algebraic Computation*, pages 216–226, London, UK, 1979. Springer-Verlag.

[85] Richard Zippel. Interpolating polynomials from their values. *J. Symb. Comput.*, 9(3):375–403, 1990.