SPARSE POLYNOMIAL INTERPOLATION AND THE FAST EUCLIDEAN ALGORITHM

by

Soo H. Go

B.Math., University of Waterloo, 2006

A Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in the
Department of Mathematics
Faculty of Science

SIMON FRASER UNIVERSITY
Summer 2012

APPROVAL

Name: Soo H. Go

Degree: Master of Science

Title of Thesis: Sparse Polynomial Interpolation and the Fast Euclidean Algorithm

Examining Committee: Dr. Stephen Choi
Chair

Dr. Michael Monagan
Senior Supervisor
Professor

Dr. Petr Lisonek
Supervisor
Associate Professor

Dr. Nils Bruin
Internal Examiner
Associate Professor

Date Approved: July 3, 2012

Abstract

We introduce an algorithm to interpolate sparse multivariate polynomials with integer coefficients. Our algorithm modifies Ben-Or and Tiwari's deterministic algorithm for interpolating over rings of characteristic zero to work modulo p, a smooth prime of our choice. We present benchmarks comparing our algorithm to Zippel's probabilistic sparse interpolation algorithm, demonstrating that our algorithm makes fewer probes for sparse polynomials.

Our interpolation algorithm requires finding roots of a polynomial in $\mathrm{GF}(p)[x]$, which in turn requires an efficient polynomial GCD algorithm. Motivated by this observation, we review the Fast Extended Euclidean algorithm for univariate polynomials, which recursively computes the GCD using a divide-and-conquer approach. We present benchmarks for our implementation of the classical and fast versions of the Euclidean algorithm demonstrating a good speedup. We discuss computing resultants as an application of the fast GCD algorithm.

To my family

"It wouldn't be inaccurate to assume that I couldn't exactly not say that it is or isn't almost partially incorrect."

— *Pinocchio,* Shrek the Third, *2007*

Acknowledgments

First of all, I am thankful for all the guidance my supervisor Dr. Michael Monagan has given me during my masters at Simon Fraser University, introducing me to computer algebra, encouraging me to not settle for good enough results, and teaching me to be enthusiastic about my work. I am also grateful to Mahdi Javadi for generously sharing with me with his code. Of course, I owe much gratitude to my family for their continued patience, love, and support. Finally, I want to thank my friends who gracefully put up with my erratic availability and constant need for technical, emotional, and moral support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, we are interested in efficient algorithms for polynomial manipulation, particularly interpolation of sparse polynomials and computing the greatest common divisor of two polynomials.

1.1 Polynomial interpolation

The process of determining the underlying polynomial from a sequence of its values is referred to as *interpolating a polynomial from its values*. Polynomial interpolation is an area of great interest due to its application in many algorithms in computer algebra that manipulate polynomials such as computing the greatest common divisor (GCD) of polynomials or the determinant of a matrix of polynomials.

Example 1.1. Let F be a field. Consider an $n \times n$ matrix M, whose entries are univariate polynomials in $F[x]$ of degrees at most d, and let $D = \det M$. Then $\deg D \leq nd$. In order to compute D, one can use the cofactor expansion, which requires $O(n2^n)$ arithmetic operations in $F[x]$ (see [13]) and can be expensive in case the coefficients of the entries and n are large.

Alternatively, we can compute the determinant using the Gaussian Elimination (GE). However, the standard form of GE requires polynomial division, so we often must work over the fraction field $F(x)$. In this approach, we need to compute the GCD of the numerator and the denominator for each fraction that appears in the computation in order to cancel common factors, and this need for the polynomial GCDs drives up the cost. To avoid

working over the fraction field, we can use the Fraction-Free GE due to Bareiss [1]. It reduces the given matrix M to an upper triangular matrix and keeps track of the determinant as it proceeds so that $D = \pm M_{nn}$. The algorithm requires $O(n^3)$ multiplications and exact divisions in $F[x]$. The degrees of the polynomials in the intermediate matrices increase as the algorithm proceeds and can be as large as $nd+(n-2)d$, so a single polynomial multiplication and division can cost up to $O(n^2d^2)$ arithmetic operations in F. The average degree of the entries is $O((n/2)d)$. Thus total cost of the Fraction-Free GE is $O(n^2d^2) \times O(n^3) = O(n^5d^2)$ arithmetic operations in F.

A nice way to compute the determinant of M is to use evaluation and interpolation. First, we evaluate the polynomial entries of M for $x = \alpha_0 \in F$ using Horner's method, the cost for which can be shown to be $n^2O(d)$ arithmetic operations in F. Next, we compute the determinant of the evaluated matrix to obtain $D(\alpha_0)$ using the GE over F, which costs $O(n^3)$ arithmetic operations in F. We repeat these steps for nd distinct points $x = \alpha_1, \ldots, \alpha_{nd} \in F$ to obtain $D(\alpha_1), \ldots, D(\alpha_{nd})$. We then interpolate $D(x)$ from $D(\alpha_0), D(\alpha_1), \ldots, D(\alpha_{nd})$, which costs $O(n^2d^2)$ arithmetic operations in F. The overall cost of the evaluate and interpolate approach is $O((nd+1)n^2d + (nd+1)n^3 + n^2d^2) = O(n^3d^2 + n^4d)$, which is an improvement of two orders of magnitude over Fraction-Free GE.

In designing an efficient algorithm for multivariate polynomial computations, it is often crucial to be mindful of the expected sparsity of the polynomial, because an approach that is efficient for dense polynomials may not be for sparse cases. Let us first clarify what sparse polynomials are.

Definition 1.2. Let R be a ring, and let $f \in R[x_1, x_2, \ldots, x_n]$. Suppose f has t nonzero terms and $\deg f = d$. The maximum possible number of terms f can have is $T_{max} = \binom{n+d}{d}$. We say f is *sparse* if $t \ll T_{max}$.

Example 1.3. Suppose $f = x_1^d + x_2^d + \cdots + x_n^d$. To interpolate f, Newton's interpolation algorithm requires $(d+1)^n$ values when f only has n nonzero terms. In contrast, Zippel's sparse interpolation algorithm requires $O(dn^2)$ values. These values are generated by evaluating the underlying polynomial and are often expensive to compute. In case of a large n, Newton's algorithm costs much more than Zippel's.

We now introduce the computation model of the interpolation problem. Let R be an arbitrary ring. *Black box* B containing a polynomial $f \in R[x_1, x_2, \ldots, x_n]$ takes in the input

$(\alpha_1, \alpha_2, \ldots, \alpha_n) \in R^n$ and returns $f(\alpha_1, \alpha_2, \ldots, \alpha_n) \in R$. The determinant of a polynomial matrix or the GCD of two polynomials can be viewed as an instance of a black box.

Figure 1.1: Blackbox representation of function f

Kaltofen and Trager introduced this model in [21], wherein the black box represents a subroutine or a program that computes $f(\alpha_1, \ldots, \alpha_n)$. The black box implicitly defines a multivariate polynomial through substituting elements from a given field for the variables. They claim that by adopting the implicit representation, the black box algorithms achieve efficiency in space over the conventional representations [14].

Example 1.4. Let M be an $n \times n$ Vandermonde matrix, where $M_{ij} = x_i^{j-1}$ and $D = \det M = \prod_{1 \le i < j \le n}(x_i - x_j) \in \mathbb{Z}[x_1, \ldots, x_n]$. The expanded polynomial D has $n!$ nonzero terms, so working with the expanded polynomial in conventional representation requires $O(n!)$ storage. On the other hand, we can use the black box to represent D as a product of irreducible factors. Since D is a product of $\sum_{i=1}^{n-1} i$ linear factors, the black box representation requires $O(n^2)$ storage.

The *sparse interpolation problem* is to determine the nonzero coefficients c_i and monomials $M_i = x_1^{e_{i1}} x_2^{e_{i2}} \cdots x_n^{e_{in}}$ so that $f = \sum_{i=1}^{t} c_i M_i, c_i \in R \backslash \{0\}$ for a given black box B with a sparse underlying polynomial f. Sometimes the interpolation algorithm will need a bound $T \ge t$ on the number of terms in f or a bound $D \ge \deg f$ on the degree of f.

Remark 1.5. Throughout this thesis, we will interchangeably use the expression evaluating f at $\alpha \in R^n$ with *probing black box B containing f*.

1.2 Polynomial GCD

Computing the GCD of two polynomials is a long studied problem with many applications. The important problem of computing the zeros of a polynomial is closely related to the problem of computing the GCD, which in turn creates a close relationship between computing GCD and the problem of factoring polynomials. (We will explore this relationship more closely in Chapter 3.) Computing with rational functions, as mentioned in Example

1.1, is an example for which the ability to efficiently compute the GCD is required in order to reduce the size of the functions by cancelling the GCD from the numerator and the denominator.

There are different approaches for computing polynomial GCDs for $\mathbb{Z}[x_1, \ldots, x_n]$. Some modern algorithms use the sparsity of the polynomials and utilize polynomial interpolation techniques to improve the cost. Brown's Modular GCD algorithm introduced in [7] solves the potential problem of large coefficients in computations by projecting down the problem to find the solution modulo a set of primes and find the GCD using the Chinese Remainder Theorem. This is an example of a a dense method. Zippel's sparse modular GCD algorithm in [34] improves upon Brown's algorithm by reducing the number of univariate images of the target polynomial using a probabilistic technique. Zippel's multivariate GCD algorithm is currently used by Maple, Magma, and Mathematica. In this thesis, we will focus on variations of the classical Euclidean algorithm for polynomial GCDs. We present a way to speed up the computations using properties of the *polynomial remainder sequence*.

Remark 1.6. Zippel's algorithm obtains univariate images of the GCD and uses interpolation. The process of generating a univariate image given an evaluation point can be viewed as a black box $B : F^{n-1} \to F[x_n]$.

Figure 1.2: Blackbox representation for generating univariate images of g

1.3 Outline

In Chapter 2, we review four polynomial algorithms, namely Newton's classical algorithm, Zippel's probabilistic algorithm [1979], Ben-Or and Tiwari's deterministic algorithm [1988], and Javadi and Monagan's algorithm [2010]. We then present a new algorithm to interpolate a sparse polynomial within a black box with coefficients over a finite field. We base our algorithm on Ben-Or and Tiwari's algorithm. Our goal is to develop an algorithm that requires as few probes to the black box as possible to interpolate sparse f with integer coefficients in polynomial time complexity in n, d, and t, where n is the number of variables, d is the degree of f, and t is the number of nonzero terms in f. The new algorithm performs

$O(t)$ probes for sparse polynomial just as the Ben-Or & Tiwari's algorithm does, which is a factor of $O(nd)$ less than Zippel's algorithm which makes $O(ndt)$ probes. We include timings for our implementation of the new algorithm on various inputs and compare them against those for Zippel's algorithm.

Ben-Or and Tiwari's sparse interpolation algorithm, as well as our new algorithm, computes the roots of a degree t polynomial $\Lambda(z)$ over the field $\mathbb{Z}_p$, which is often cost-intensive. The bottleneck of the root finding process is computing a series of polynomial GCDs to identify products of linear factors of $\Lambda(z)$. Motivated by this observation, we review a fast GCD algorithm in Chapter 3. The root finding process requires $O(t^2 \log p)$ field operations when using the classical Euclidean algorithm. However, implementing the fast Euclidean algorithm reduces the complexity to $O(t \log t \log tp)$. We begin the chapter by describing the naive versions of the Euclidean algorithm that computes the GCD of two elements of a Euclidean domain. Next, we examine the Fast Extended Euclidean Algorithm (FEEA) [11] for computing polynomial GCD. The fast algorithm uses a recursive process to efficiently compute the GCD of two polynomials both of degree at most n in $F[x]$ for some field F in $O(M(n) \log n)$ arithmetic operations in F, where $M(n)$ is the number of arithmetic operations required to multiply two polynomials of degree at most n. In our implementation, we use the Karatsuba polynomial multiplication algorithm, so $M(n) = O(n^{\log_2 3}) \approx O(n^{1.58})$. Implementing the FFT would bring $M(n)$ down further to $O(n \log n)$. We present timings from our implementation of the algorithms to demonstrate the savings in running time from using the FEEA over the traditional Euclidean algorithm. Finally, as another application of the FEEA, we modify it to compute the resultant of two polynomials in $F[x]$ in $O(M(n) \log n)$ arithmetic operations in F.

Chapter 2

Sparse Polynomial Interpolation

In this section, we focus on techniques for interpolating sparse polynomials. First, we review four known algorithms, the first of which is the classical algorithm due to Newton. This is a dense algorithm, in that the number of evaluations required depends solely on the degree bound and the number of variables regardless of the number of nonzero terms in the target polynomial. The next three algorithms of Zippel [34], Ben-Or and Tiwari [3], and Javadi and Monagan [16] were created for sparse polynomials. Finally, we introduce a new sparse interpolation algorithm which performs computations over a finite field. We discuss the runtime cost of the new algorithm and present timings of our implementation of the algorithm, which is compared against the algorithms by Zippel and Javadi and Monagan.

2.1 Previous Works

2.1.1 Newton's Algorithm

Let F be a field and $f \in F[x]$. Newton and Lagrange's interpolation algorithms are classical methods for interpolating f. They are examples of *dense interpolation* and are inefficient for sparse polynomials due to the number of evaluations they make, as observed in Example 1.3 with Newton interpolation.

Lagrange interpolation uses the *Lagrange interpolation formula*, which for a degree d univariate target polynomial $f(x) = c_0 + c_1x + \cdots + c_dx^d$ with evaluation points $\alpha_0, \ldots, \alpha_d$

is

$$f_j(x) = \prod_{\substack{i \neq j \\ 0 \leq i \leq d}} \frac{x - \alpha_i}{\alpha_j - \alpha_i}, \quad 0 \leq j \leq d,$$

so that $f(x) = \sum_{0 \leq j \leq d} f_j(x) v_j$, where $v_j = f(\alpha_j)$, $0 \leq j \leq d$.

Newton's interpolation algorithm first expresses the univariate solution of degree d in the form

$$f(x) = v_0 + v_1(x - \alpha_0) + v_2(x - \alpha_0)(x - \alpha_1) + \cdots + v_d \prod_{i=0}^{d-1} (x - \alpha_i), \tag{2.1.1}$$

where the *evaluation points* $\alpha_0, \alpha_1, \ldots \alpha_d \in F$ are pairwise distinct and the *Newton coefficients* $v_1, v_2, \ldots, v_d \in F$ are unknown. Then the coefficients v_i are determined by

$$v_i = \begin{cases} f(\alpha_0), & i = 0 \\ \left(f(\alpha_i) - [v_0 + \cdots + v_{i-1} \prod_{k=0}^{i-2} (\alpha_i - \alpha_k)]\right) \left(\prod_{k=0}^{i-1} (\alpha_i - \alpha_k)\right)^{-1}, & i = 1, \ldots, d. \end{cases} \tag{2.1.2}$$

Note that computing $f_{j+1}(x)$ in the Lagrange interpolation does not utilize the previously determined $f_0(x), \ldots, f_j(x)$, whereas the Newton interpolation algorithm determines the target polynomial by building directly upon the results from the previous steps.

In the univariate case, both algorithms require $d+1$ points and require $O(d^2)$ arithmetic operations in F to interpolate a polynomial of degree at most d. ([36], Chapter 13).

For a multivariate polynomial f in $x_1, x_2, \ldots, x_n$, Newton's algorithm proceeds by interpolating for one variable at a time with v_i and $f(\alpha_i)$, which are multivariate polynomials themselves. We illustrate the multivariate version of Newton's interpolation algorithm with the following example.

Example 2.1. Suppose $F = \mathbb{Z}_7$ and that we are given a black box with an underlying polynomial $f(x, y) = x^2 y + 5y + 1 \in \mathbb{Z}_7[x, y]$ and know that $\deg_x f = 2$ and $\deg_y f = 1$. First, we fix $x = 0$ and interpolate in y. That is, we will interpolate $f(0, y)$. Since $\deg_y f = 1$, we need two evaluations. So, we take $\alpha_0 = 0, \alpha_1 = 1$ and get $f(0, \alpha_0) = 1, f(0, \alpha_1) = 6$ from the black box. Using (2.1.2), we compute $v_0 = 1$ and $v_1 = 5$. We substitute these values into (2.1.1) and find

$$f(0, y) = 1 + 5(y - 0) = 5y + 1.$$

This $f(0, y) \in \mathbb{Z}_7[y]$ constitutes a single evaluation of $f(x, y)$ at $x = 0$.

Now, let $f(x, y) = a_1(x)y + a_0(x)$. Since $\deg_x f = 2$, we need two more images of $f(x, y)$ in $\mathbb{Z}[y]$ to determine $a_0(x)$ and $a_1(x)$. Repeating the process of interpolating f in y with $x = 1$ and $x = 2$, we obtain the following result.

α_i	$f(\alpha_i, y)$
0	$\boxed{5}y+\boxed{1}$
1	$\boxed{6}y+\boxed{1}$
2	$\boxed{2}y+\boxed{1}$

Table 2.1: Univariate images of f

At this point, we can take $\{a_1(0) = 5, a_1(1) = 6, a_1(2) = 2\}$ and interpolate in x to obtain $a_1(x) = x^2 + 5$. Likewise, we find $a_0(x) = 1$ with the constant terms of $f(\alpha_i, y)$. Finally, we have

$$f(x, y) = (x^2 + 5)y + 1 = x^2y + 5y + 1,$$

which is the desired polynomial in the domain $\mathbb{Z}_7[x, y]$.

Remark 2.2. The total number of evaluation points needed to interpolate a multivariate polynomial f using the Newton interpolation algorithm is

$$\prod_{i=1}^{n}(d_i + 1) < (d + 1)^n, \tag{2.1.3}$$

for some degree bound d, $d \geq d_i = \deg_{x_i} f, i = 1, 2, \ldots, n$. This is exponential in n and will grow very quickly as n and d increase, which is inefficient for sparse polynomials.

2.1.2 Zippel's Sparse Interpolation Algorithm

Zippel's multivariate polynomial interpolation algorithm is probabilistic with expected running time that is polynomial in the number of terms in f. The solution is built up by interpolating one variable at a time: First, the *structure* (or form) of the polynomial is determined by using dense interpolation such as Newton's algorithm. This structure is used as the basis for generating values for a series of sparse interpolations.

Before we present more details of Zippel's algorithm, we consider the following lemma.

Lemma 2.3. *(Schwartz [31]) Let K be a field and $f \in K[x_1, x_2, \ldots, x_n]$ nonzero. Let $d = \deg f$. Let S be a finite subset of K, and let $r_1, \ldots, r_n$ be random evaluation points*

chosen from S. Then

$$\Pr(f(r_1, r_2, \ldots, r_n) = 0) \leq \frac{d}{|S|}.$$

In particular, if $K = \mathbb{Z}_p$ for some prime p, $\Pr(f(\alpha) = 0) \leq \frac{d}{p}$ for a randomly chosen $\alpha \in \mathbb{Z}_p^n$.

Remark 2.4. The above lemma is commonly known as the Schwartz-Zippel Lemma. There is some controversy as to who was the first to produce this result, as there are up to three possible sources, Schwartz [31], Zippel [34], and DeMillo and Lipton. The first to be published was DeMillo and Lipton in 1978, and unaware of this work, Schwartz and Zippel presented their independent results in 1979. ([36], Chapter 12)

The key idea of Zippel's algorithm lies in the assumption that at every stage of interpolating in x_i and updating the skeleton of the target polynomial, the first image of f is computed with a *good starting point*. Consider the structure of f after k steps of the algorithm, which can be written as

$$f(x_1, \ldots, x_n) = f_{1,k}(x_{k+1}, \ldots, x_n)x_1^{e_{11}} \cdots x_k^{e_{1k}} + \cdots + f_{t,k}(x_{k+1}, \ldots, x_n)x_1^{e_{t1}} \cdots x_k^{e_{tk}}.$$

If some $f_{i,k}$ vanishes at the starting evaluation point of the next step, then the new structure produced is strictly smaller than it should be and the interpolation fails. On the other hand, if none of $f_{i,k}$ evaluates to zero at the starting point, the new structure is a correct image of the form of f. Fortunately, the probability of any $f_{i,k}$ vanishing at a point is small for a large enough p, as shown in Lemma 2.3, provided that the evaluation point α is chosen at random. (Zero, as I had the misfortune to learn firsthand, is not the best choice out there.)

We will describe the algorithm in more detail through the following example.

Example 2.5. Let $p = 17$. Suppose we are given a black box that represents the polynomial $f = x^5 - 7x^3y^2 + 2x^3 + 6yz - z + 3 \in \mathbb{Z}_p[x, y, z]$. Suppose further we somehow know $d_x = \deg_x f = 5, d_y = \deg_y f = 2$, and $d_z = \deg_z f = 1$.

We begin by choosing at random $\beta_0, \gamma_0 \in \mathbb{Z}_p$ and interpolating for x to find $f(x, \beta_0, \gamma_0)$ by making $d_x + 1 = 6$ probes to the black box. Suppose $\beta_0 = 2$ and $\gamma_0 = -6$. We evaluate $f(\alpha_i, \beta_0, \gamma_0) \bmod p$ with $\alpha_i \in \mathbb{Z}_p$ chosen at random for $i = 0, 1, \ldots, d_x = 5$ to find $f(x, 2, -6) \bmod p$. Suppose we have $\alpha_0 = 0, \alpha_1 = 1, \ldots, \alpha_5 = 5$. Then we have

$$\begin{aligned} &f(0, 2, -6) = 5; \quad f(1, 2, -6) = -3; \quad f(2, 2, -6) = -1; \\ &f(3, 2, -6) = 5; \quad f(4, 2, -6) = -6; \quad f(5, 2, -6) = -1. \end{aligned}$$

With these six points, we use a dense interpolation algorithm such as the Newton or Lagrange's algorithm to find

$$f(x, 2, -6) = x^5 + 0x^4 + 8x^3 + 0x^2 + 0x + 5.$$

The next step shows how the probabilistic assumption of Zippel's algorithm is used to find the structure of f. We assume that if some power of x had a zero coefficient in $f(x, \beta_0, \gamma_0)$, it will have a zero coefficient in $f(x, y, z)$ as well. That is, there is a high probability that the target polynomial is $f(x, y, z) = a_5(y, z)x^5 + a_3(y, z)x^3 + a_0(y, z)$ for some $a_5, a_3, a_0 \in \mathbb{Z}_p[y, z]$.

The algorithm proceeds to interpolate each of the three coefficients for variable y. Since $d_y = 2$ in this example, we need two more images of f to interpolate for variable y. Pick β_1 from $\mathbb{Z}_p$ at random. We find $f(x, \beta_1, \gamma_0)$ by interpolating, and the only coefficients we need to determine are the nonzero ones, namely $a_5(\beta_1, \gamma_0), a_3(\beta_1, \gamma_0)$, and $a_0(\beta_1, \gamma_0)$ in this example, since we expect that the other ones are identically zero. Note that in general, we have at most t of these unknown coefficients.

We can find the coefficients by solving a system of linear equations of size at most $t \times t$. Here, we have three nonzero unknown coefficients, so we need three evaluations for the system of equations instead of the maximum $t = 6$. Suppose $\beta_1 = 3$. Then three new evaluations give

$$\begin{aligned}
f(\alpha_0, \beta_1, \gamma_0) = &\quad 0a_5(\beta_1, \gamma_0) + 0a_3(\beta_1, \gamma_0) + a_0(\beta_1, \gamma_0) = 3, \\
f(\alpha_1, \beta_1, \gamma_0) = &\quad a_5(\beta_1, \gamma_0) + a_3(\beta_1, \gamma_0) + a_0(\beta_1, \gamma_0) = -6, \\
f(\alpha_2, \beta_1, \gamma_0) = &-2a_5(\beta_1, \gamma_0) + 8a_3(\beta_1, \gamma_0) + a_0(\beta_1, \gamma_0) = 6.
\end{aligned}$$

Solving this system of equations shows $a_5(\beta_1, \gamma_0) = 1, a_3(\beta_1, \gamma_0) = 7$, and $a_0(\beta_1, \gamma_0) = 3$. Hence

$$f(x, 3, -6) = x^5 + 7x^3 + 3.$$

Note that there is a chance the linear equations developed in this step are linearly dependent if random $\alpha_i, \beta_j, \gamma_k \in \mathbb{Z}_p$ are used, in which case the resulting system is singular. If so, we evaluate at more points until a linearly independent system is formed. As well, consider the case where our zero coefficient assumptions are incorrect in some step in the algorithm so that some nonzero coefficient is assumed to be zero. The final solution will be incorrect, and so the algorithm fails. Thus we need to check our assumptions about the zero coefficients are true throughout the interpolation process. We can check for the correctness

of the assumptions made in the previous steps by using $d+2$ evaluation points instead of $d+1$. The extra evaluations will detect with a high probability the unlucky assumption by making the system inconsistent.

So far, we have two images $f(x, \beta_0, \gamma_0)$ and $f(x, \beta_1, \gamma_0)$, so we proceed to generate the third image. We repeat the process of picking $y = \beta_2$ at random, setting up a linear system and solving the system to find the image $f(x, \beta_2, \gamma_0)$. Suppose $\beta_2 = 7$. We obtain $f(x, 7, -6) = x^5 - x^3 - 5$.

Now, we interpolate each coefficient for y: From

$$\{a_3(2, -6) = 2, a_3(3, -6) = 7, a_3(7, -6) = -1\},$$

we compute $a_3(y, -6) = -7y^2+2$. We determine in the same manner $a_5(y, -6)$ and $a_0(y, -6)$ using the other two sets of coefficients and see

$$\begin{aligned} a_5(y, \gamma_0) &= 1, \\ a_3(y, \gamma_0) &= -7y^2 + 2, \\ a_0(y, \gamma_0) &= -2y - 8, \end{aligned}$$

from which we get

$$f(x, y, -6) = x^5 - 7x^3y^2 + 2x^3 - 2y - 8.$$

Next, we proceed again with the probabilistic sparse interpolation for variable z. First, we update the form of the solution:

$$f(x, y, z) = b_{5,0}(z)x^5 + b_{3,2}(z)x^3y^2 + b_{3,0}(z)x^3 + b_{0,1}(z)y + b_{0,0}(z),$$

where $b_{5,0}, b_{3,2}, b_{3,0}, b_{0,1}, b_{0,0} \in \mathbb{Z}_p[z]$. There are five terms in this form, so we need five evaluations to set up the system of linear equations as before. Let $\gamma_1 = -4$. Then by evaluating $f(\lambda_i, \theta_i, \gamma_1)$ for $i = 0, 1, \ldots, 4$, where $(\lambda_i, \theta_i) \in \mathbb{Z}_p^2$ are chosen at random, and solving the resulting system of equations, we see

$$b_{5,0}(\gamma_1) = 1, b_{3,2}(\gamma_1) = -7, b_{3,0}(\gamma_1) = 2, b_{0,1}(\gamma_1) = -7, b_{0,0}(\gamma_1) = 7$$

and thus

$$f(x, y, -4) = x^5 - 7x^3y^2 + 2x^3 - 7y + 7.$$

Interpolating for z using the coefficients from $f(x, y, -6)$ and $f(x, y, -4)$, we finally obtain

$$f(x, y, z) = x^5 - 7x^3y^2 + 2x^3 + 6yz - z + 3.$$

Zippel's algorithm makes 17 probes to the black box for Example 2.5. Newton's interpolation algorithm would have made 36 probes given the same black box. In general, if all probabilistic assumptions hold, Zippel's algorithm makes $O(ndt)$ probes to the black box for some degree bound d.

2.1.3 Ben-Or and Tiwari's Sparse Interpolation Algorithm

Ben-Or and Tiwari's interpolation algorithm for multivariate sparse polynomials over rings with characteristic zero is a *deterministic algorithm*, in that it does not use randomization. Assume $f(x_1, \ldots, x_n) \in \mathbb{Z}[x_1, \ldots, x_n]$ has t nonzero terms. In contrast to the last two algorithms, Ben-Or and Tiwari's algorithm does not interpolate a multivariate polynomial one variable at a time. To run Ben-Or and Tiwari's algorithm, we need a term bound $T \geq t$, which is an extra requirement over that for Zippel's algorithm. On the other hand, Ben-Or and Tiwari's algorithm does not require a bound on the partial degrees $d_i = \deg_{x_i} f$, $1 \leq i \leq n$.

Write $f = c_1 M_1 + c_2 M_2 + \cdots + c_t M_t$, where $M_i = x_1^{e_{i1}} x_2^{e_{i2}} \cdots x_n^{e_{in}}$ are the monomials of f and the exponents e_{ij} and the nonzero coefficients c_i are unknown. Given the number of variables n and the term bound $T \geq t$, Ben-Or and Tiwari's algorithm uses the first n primes $p_1 = 2, p_2 = 3, \ldots, p_n$ in the $2T$ evaluation points $\alpha_i = (2^i, 3^i, \ldots, p_n^i)$, $0 \leq i \leq 2T-1$. The algorithm can be divided into two phases. In the first phase, we determine e_{ij} using a *linear generator*, and then in the second phase we determine c_i by solving a linear system of equations over $\mathbb{Q}$.

We introduce here a direct way to find the linear generator. Suppose for simplicity $T = t$. (We will deal with the case $T > t$ later.) Let v_i be the output from a probe to the black box with the input α_i, i.e., $v_i = f(\alpha_i)$ for $0 \leq i \leq 2t-1$, and let $m_j = M_j(\alpha_1)$ for $1 \leq j \leq t$. The linear generator is defined to be the monic univariate polynomial $\Lambda(z) = \prod_{i=1}^{t}(z - m_i)$, which when expanded forms $\Lambda(z) = \sum_{i=0}^{t} \lambda_i z^i$ with $\lambda_t = 1$. Once the coefficients λ_i are found, we compute all integer roots of $\Lambda(z)$ to obtain m_i.

We find λ_i by creating and solving a linear system as follows:
Since $\Lambda(m_i) = 0$ for any $1 \leq i \leq t$, we have

$$0 = c_i m_i^l \Lambda(m_i) = c_i(\lambda_0 m_i^l + \lambda_1 m_i^{l+1} + \cdots + \lambda_t m_i^{t+l}).$$

Summing over i gives

$$0 = \lambda_0 \sum_{i=1}^{t} c_i m_i^l + \lambda_1 \sum_{i=1}^{t} c_i m_i^{1+l} + \cdots + \lambda_t \sum_{i=1}^{t} c_i m_i^{t+l}. \tag{2.1.4}$$

Note

$$\begin{aligned} m_i^l &= 2^{e_{i1}l} 3^{e_{i2}l} \cdots p_n^{e_{in}l} \\ &= (2^l)^{e_{i1}} (3^l)^{e_{i2}} \cdots (p_n^l)^{e_{in}} \\ &= M_i(2^l, 3^l, \ldots, p_n^l) \\ &= M_i(\alpha_l), \end{aligned} \tag{2.1.5}$$

so

$$\sum_{i=1}^{t} c_i m_i^l = \sum_{i=1}^{t} c_i M_i(\alpha_l) = f(\alpha_l) = v_i.$$

Then from (2.1.4), we have

$$0 = \lambda_0 v_l + \lambda_1 v_{1+l} + \cdots + \lambda_t v_{t+l} \Longleftrightarrow \lambda_0 v_l + \lambda_1 v_{1+l} + \cdots + \lambda_{t-1} v_{t-1+l} = -\lambda_t v_{t+l}.$$

Recall also that $\Lambda(z)$ is monic and so $\lambda_t = 1$. Hence we have

$$\lambda_0 v_l + \lambda_1 v_{1+l} + \cdots + \lambda_{t-1} v_{t+l-2} + v_{t+l-1} = -v_{t+l}.$$

Since we need to determine t coefficients $\lambda_0, \lambda_1, \ldots, \lambda_{t-1}$, we need t such linear relations. Letting $l = 0, 1, \ldots, t-1$, we see we need $v_0, v_1, \ldots, v_{2t-1}$, so we make $2t$ probes to the black box. Once we have these evaluations, we can solve $V\bar{\lambda} = -\bar{v}$, where

$$V = \begin{bmatrix} v_0 & v_1 & \cdots & v_{t-1} \\ v_1 & v_2 & \cdots & v_t \\ \vdots & \vdots & \ddots & \vdots \\ v_{t-1} & v_t & \cdots & v_{2t-2} \end{bmatrix}, \quad \bar{\lambda} = \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_{t-1} \end{bmatrix}, \quad \text{and} \quad -\bar{v} = \begin{bmatrix} -v_t \\ -v_{t+1} \\ \vdots \\ -v_{2t-1} \end{bmatrix}. \tag{2.1.6}$$

We had assumed until now $T = t$. If we are given $T > t$, we can find t by computing $\operatorname{rank}(V)$, correctness of which comes from the following theorem and its corollary.

Theorem 2.6 ([3], Section 4)**.** *Let V_l denote the square matrix consisting of the first l rows and columns of V. If t is the exact number of monomials appearing in f, then*

$$\det V_l = \begin{cases} \displaystyle\sum_{S \subset \{1,2,\ldots,t\}, |S|=l} \left(\prod_{i \in S} c_i \prod_{i>j, i,j \in S} (m_i - m_j)^2 \right) & l \leq t \\ 0 & l > t. \end{cases}$$

Corollary 2.7 ([3], Section 4)**.** *If the number of nonzero coefficients in f is bounded by T, then the number of nonzero coefficients in f equals* $\max_{V_j \text{ is nonsingular}, j \leq T} \{j\}$.

The above method of finding a linear generator $\Lambda(z)$ for f costs $O(T^3)$ arithmetic operations in $\mathbb{Q}$. In our implementation, we use the Berlekamp-Massey process [25], a technique from coding theory, to reduce the runtime to $O(T^2)$ arithmetic operations in $\mathbb{Q}$.

Next, we need to find the integer roots of $\Lambda(z)$, which can be achieved in $O(t^3 dn \log n)$ operations, where $d = \deg f$, using a Hensel lifting based p-adic root finding algorithm by Loos (1983). Once we have found m_i, we use the fact $m_i = M_i(\alpha_1) = 2^{e_{i1}} 3^{e_{i2}} \cdots p_n^{e_{in}}$ and find exponents e_{ij} by factoring each of m_i into products of primes, which can be done by trial divisions.

Finally, the coefficients c_i are found easily by solving the $t \times t$ system of linear equations

$$c_1 M_1(\alpha_i) + \cdots + c_t M_t(\alpha_i) = v_i, \ 0 \leq i \leq t-1. \tag{2.1.7}$$

Recall from (2.1.5) that $M_j(\alpha_i) = m_j^i$. Thus the system in (2.1.7) can be written as $A\bar{c} = \bar{v}$, where

$$A = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ m_1^1 & m_2^1 & \cdots & m_t^1 \\ \vdots & \vdots & \ddots & \vdots \\ m_1^{t-1} & m_2^{t-1} & \cdots & m_t^{t-1} \end{bmatrix}, \quad \bar{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_t \end{bmatrix}, \quad \text{and} \quad \bar{v} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{t-1} \end{bmatrix}. \tag{2.1.8}$$

Solving the system above is easy, because A is a transposed Vandermonde matrix. Inverting A can be done in $O(t^2)$ time complexity [18].

We demonstrate the algorithm in the following example.

Example 2.8. Suppose we are given a black box B representing $f = 3x^5 - 5xy^2 + 2y^2z + 6$ along with term bound $T = t = 4$. We will use three primes $p_1 = 2, p_2 = 3$, and $p_3 = 5$. Evaluating f at $2T = 8$ evaluation points $\alpha_i = (2^i, 3^i, 5^i)$, $i = 0, \ldots, 7$ gives $\{v_0 = 6; v_1 = 102; v_2 = 5508; v_3 = 251400; v_4 = 10822104; v_5 = 460271712; v_6 = 19658695608; v_7 = 847357021200\}$.

The corresponding linear generator is

$$\Lambda(z) = z^4 - 96z^3 + 2921z^2 - 28746z + 25920,$$

whose integer roots are

$$R = \{45 = 3^2 \times 5^1, 1, 32 = 2^5, 18 = 2 \times 3^2\}.$$

Hence we can deduce

$$M_1 = y^2z; M_2 = 1; M_3 = x^5; M_4 = xy^2.$$

To compute the coefficients c_i, we solve the system of linear equations

$$c_1M_1(\alpha_i) + \cdots + c_4M_4(\alpha_i) = v_i,\ 0 \leq i \leq 3.$$

We find $c_1 = 2, c_2 = 6, c_3 = 3$, and $c_4 = -5$. Putting together the monomials and the coefficients, we find the correct target polynomial

$$f(x, y, z) = 3x^5 - 5xy^2 + 2y^2z + 6.$$

Definition 2.9. An interpolation algorithm is *nonadaptive* if it determines all of the evaluation points based solely on the given bound, T, on the number of monomials.

Definition 2.10. Let f be a polynomial with at most T distinct monomials. Then f is called *T-sparse*.

Theorem 2.11 ([3], Section 7)**.** *Any nonadaptive polynomial interpolation algorithm which determines a T-sparse polynomial in n variables must perform at least $2T$ evaluations.*

Proof. The proof of the theorem is based on the observation that every nonzero l-sparse polynomial f, where $l < 2T$, can be rewritten as the sum of two distinct T-sparse polynomials. We will show the proof of the univariate case here. Suppose the interpolation algorithm chooses $l < 2T$ points $\alpha_1, \ldots, \alpha_l$ for the given T-sparse polynomial to evaluate. Construct the polynomial

$$p(x) = \prod_{i=1}^{l}(x - \alpha_i) = \sum_{i=0}^{l} c_i x^i,$$

and let

$$p_1(x) = \sum_{i=0}^{\lfloor l/2 \rfloor} c_i x^i \quad \text{and} \quad p_2(x) = -\sum_{i=\lfloor l/2 \rfloor + 1}^{l} c_i x^i.$$

Since $p(x)$ has at most $l + 1 \leq 2T$ nonzero coefficients, $p_1(x)$ and $p_2(x)$ are both T-sparse. Moreover, by construction, $p(\alpha_i) = p_1(\alpha_i) - p_2(\alpha_i) = 0$ for $1 \leq i \leq l$, so $p_1(\alpha_i) = p_2(\alpha_i)$ for all $1 \leq i \leq l$. We need an additional evaluation point α for which $p(\alpha) \neq 0$ so that $p_1(\alpha) \neq p_2(\alpha)$. That is, for any set of $l < 2T$ evaluation points, we can construct two distinct T-sparse polynomials that return the identical sets of outputs and require at least one more

additional evaluation. Therefore, $2T$ is a lower bound for the number of evaluations needed to interpolate a T-sparse polynomial.

□

Theorem 2.11 shows that Ben-Or and Tiwari's interpolation algorithm makes the minimum possible number of probes with the given bound T on the number of terms given the algorithm's nonadaptive approach to the interpolation problem.

Remark 2.12. The size of the evaluations $f(2^i, 3^i, \ldots, p_n^i)$, $i = 0, 1, \ldots, 2T-1$ can be as big as $p_n^{d(2T-1)}$, which is $O(Td \log p_n)$ bits long. As the parameters grow, this bound on the output grows quickly, making the algorithm very expensive in practice and thus not very useful.

2.1.4 Javadi and Monagan's Parallel Sparse Interpolation Algorithm

A drawback of Ben-Or and Tiwari's algorithm is that it cannot be used in modular algorithms such as GCD computations modulo prime p, where p is chosen to be a machine prime, as the prime decomposition step ($m_i = 2^{e_{i1}} 3^{e_{i2}} \cdots p_n^{e_{in}}$) does not work over a finite field. The parallel sparse interpolation algorithm due to Javadi and Monagan [17] modifies Ben-Or and Tiwari's algorithm to interpolate polynomials over $\mathbb{Z}_p$. Given a t-sparse polynomial, the algorithm makes $O(t)$ probes to the black box for each of the n variables for a the total of $O(nt)$ probes. The cost increase incurred by the extra factor of $O(n)$ number of probes is offset by the speedup in overall runtime from the use of parallelism.

Let $f = \sum_{i=1}^{t} c_i M_i \in \mathbb{Z}_p[x_1, \ldots, x_n]$, where p is a prime, $c_i \in \mathbb{Z}_p \backslash \{0\}$ are the coefficients, and $M_i = x_1^{e_{i1}} x_2^{e_{i2}} \cdots x_n^{e_{in}}$ are the pairwise distinct monomials of f. Let $D \geq d = \deg f$ and $T \geq t$ be bounds on the degree and the number of nonzero terms of f. Initially, the algorithm proceeds identically to Ben-Or and Tiwari's algorithm, except instead of first n integer primes, randomly chosen nonzero $\alpha_1, \ldots, \alpha_n \in \mathbb{Z}_p$ are used for the input points. The algorithm probes the black box to obtain $v_i = f(\alpha_1^i, \ldots, \alpha_n^i)$ for $0 \leq i \leq 2T-1$ and uses the Berlekamp-Massey algorithm to generate the linear generator $\Lambda_1(z)$, whose roots are $R_1 = \{r_1, \ldots, r_t\}$, where $r_i \equiv M_i(\alpha_1, \ldots, \alpha_n) \bmod p$ for $1 \leq i \leq t$.

Now follows the main body of the algorithm. To determine the degrees of the monomials in the variable x_j, $1 \leq j \leq n$, the algorithm repeats the initial steps with new input points $(\alpha_1^i, \ldots, \alpha_{j-1}^i, \beta_j^i, \alpha_{j+1}^i, \ldots, \alpha_n^i)$, $0 \leq i \leq 2T-1$, where β_j is a new value chosen at random and $\beta_j \neq \alpha_j$. Thus α_j is replaced with β_j to generate $\Lambda_{j+1}(z)$, whose roots are $R_{j+1} =$

$\{\bar{r}_1, \ldots, \bar{r}_t\}$ with $\bar{r}_k \equiv M_i(\alpha_1, \ldots, \alpha_{j-1}, \beta_j, \alpha_{j+1}, \ldots, \alpha_n) \bmod p$ for some $1 \leq i, k \leq t$. The algorithm uses bipartite matching to determine which r_i and $\bar{r}_k$ are the roots corresponding to the monomial M_i. Then $e_{ij} = \deg_{x_j} M_i$ is determined using the fact $\bar{r}_k / r_i = (\beta_j/\alpha_j)^{e_{ij}}$ and thus $\bar{r}_k = r_i \, (\beta_j/\alpha_j)^{e_{ij}}$ is a root of $\Lambda_{j+1}(z)$: since $0 \leq e_{ij} \leq D$, we try $e_{ij} = 0, 1, \ldots, D$ until $\Lambda_{j+1}(r_i \, (\beta_j/\alpha_j)^{e_{ij}}) = 0$ while maintaining $\sum_{j=1}^{n} e_{ij} \leq D$. This process of computing $\Lambda_{j+1}(z)$ and its roots and then determining the degrees of x_j can be parallelized to optimize the overall runtime.

The coefficients c_i can be obtained from solving one system of linear equations

$$v_i = c_1 r_1^i + c_2 r_2^i + \cdots + c_t r_t^i, \text{ for } 0 \leq i \leq t-1,$$

where v_i are the black box output values used for $\Lambda_1(z)$ and r_i are the roots of $\Lambda_1(z)$ as in Ben-Or and Tiwari's algorithm.

Note that this algorithm is probabilistic. If $M_i(\alpha_1, \ldots, \alpha_n) = M_j(\alpha_1, \ldots, \alpha_n)$ for some $1 \leq i \neq j \leq t$, then $\deg \Lambda_1(z) < t$. Since there will be fewer than t roots of $\Lambda_1(z)$, the algorithm fails to correctly identify the t monomials of f. Likewise, the algorithm requires $\deg \Lambda_{k+1}(z) = t$ for all $1 \leq k \leq n$ so that the bipartite matching of the roots can be found. The algorithm guarantees that the monomial evaluations will be distinct with high probability, by requiring choosing $p \gg t^2$. To check if the output is correct, the algorithm picks one more point $\alpha \in \mathbb{Z}_p^n$ at random and tests if $B(\alpha) = f(\alpha)$. If $B(\alpha) \neq f(\alpha)$, then we know the output is incorrect. Otherwise, it is correct with probability at least $1 - \frac{d}{p}$.

2.2 A Method Using Discrete Logarithms

Let p be a prime, and let $B : \mathbb{Z}^n \to \mathbb{Z}$ be a black box that represents an unknown sparse multivariate polynomial $f \in \mathbb{Z}[x_1, x_2, \ldots, x_n] \backslash \{0\}$ with t nonzero coefficients. We can write

$$f = \sum_{i=1}^{t} c_i M_i, \text{ where } M_i = \prod_{j=1}^{n} x_j^{e_{ij}} \text{ and } c_i \in \mathbb{Z} \backslash \{0\}.$$

Our goal is to efficiently find f by determining the monomials M_i and the coefficients c_i using values obtained by probing B.

In general, probing the black box is a very expensive operation, and making a large number of probes can create a bottleneck in the interpolation process. Therefore, if we can reduce the number of probes made during interpolation, we can significantly improve the

running time of the computation for sparse f. In Example 1.3, Newton's algorithm required $(d+1)^n$ evaluations for the sparse polynomial $x_1^d + x_2^d + \cdots + x_n^d$. For large d, we have $t = n \ll (d+1)^n = (d+1)^t$, so Ben-Or and Tiwari's algorithm that makes $2T \in O(t)$ probes to B given a term bound $T \in O(t)$ is much more efficient than Newton's algorithm.

One important factor to consider in designing an algorithm is feasibility of implementation. When implementing sparse polynomial interpolation algorithm, it is often necessary to work with machine integer limitations.

Example 2.13. Recall from Remark 2.12 that the output from B generated by Ben-Or and Tiwari's algorithm may be as large as $p_n^{d(2T-1)}$, where p_n is the n-th prime. Thus, as the parameters grow, the size of the output increase rapidly, past the machine integer limits.

One approach would be to work modulo a prime p. If $p > p_n^d$, we can use Ben-Or and Tiwari's algorithm directly. However, p_n^d can be very large. For example, if $n = 10$ and $d = 100$, p_n^d has 146 digits.

Kaltofen *et al.* in [19] present a modular algorithm that addresses the intermediate number growth problem in Ben-Or and Tiwari's algorithm by modifying the algorithm to work over p^k, where p is a prime and p^k is *sufficiently large*. In particular, $p^k > p_n^d$. However, p_n^d, again, can be very large, so this approach does not solve the size problem.

Our algorithm for sparse interpolation over finite field $\mathbb{Z}_p$ is a different modification of Ben-Or and Tiwari's approach, wherein we select a prime $p > (d+1)^n$ and perform all operations over $\mathbb{Z}_p$. If $p < 2^{31}$ or $p < 2^{63}$, we can avoid potential integer overflow problems in common computing situations: numbers that arise in the computation processes that are too large for the computer hardware are reduced to fit within 32 or 64 bits.

Note that our algorithm returns f_p, where $f_p \equiv f \pmod{p}$. If $|c_i| < \frac{p}{2}$ for all i, then we have $f_p = f$. Otherwise, we interpolate f modulo more primes and apply the Chinese Remainder algorithm to recover the integer coefficients c_i. We will describe how to obtain these additional images in more detail in Remark 2.21.

2.2.1 Discrete Logs

Adapting the integer algorithm to work over $\mathbb{Z}_p$ presents challenges in retrieving the monomials, as we are no longer able to use the prime decomposition of the roots of the linear generator to determine the exponents of each variable in the monomials. We address this problem by using *discrete logarithms*.

Definition 2.14. Let $\alpha, \beta \in G = \langle\alpha\rangle$, where G is a cyclic group of order n. Given $\beta \in \langle\alpha\rangle$, the *discrete logarithm problem* is to find the unique exponent a, $0 \leq a \leq n-1$, such that $\alpha^a = \beta$. The integer a is denoted $a = \log_\alpha \beta$.

Let G be a cyclic group of order n and $\alpha, \beta \in G$, where $G = \langle\alpha\rangle$. An obvious way to compute the discrete log is to compute all powers of α until $\beta = \alpha^a$ is found. This can be achieved in $O(n)$ multiplications in G and $O(1)$ space by saving the previous result α^{i-1} to compute $\alpha^i = \alpha \cdot \alpha^{i-1}$.

The discrete log algorithm due to Shanks [32], also known as Shanks' baby-step giant-step algorithm, makes a time-memory tradeoff to improve the runtime. Let $m = \lceil\sqrt{n}\rceil$. Shanks' algorithm assembles a sorted table of precomputed $O(m)$ pairs (j, α^j) for $0 \leq j < m$ and uses a binary search to find i such that $\beta(\alpha^{-m})^i = \alpha^j \pmod p$, $0 \leq i < m$. The algorithm then returns $a = im + j$. By making clever precomputations and using a fast sorting algorithm, Shanks' algorithm solves the discrete logarithm problem in $O(m)$ multiplications and $O(m)$ memory. A detailed description of the algorithm is presented as Algorithm 6.1 in [33].

The Pohlig-Hellman algorithm [28], first expresses n as a product of distinct primes so that $n = \prod_{i=1}^k p_i^{c_i}$. Next, we compute $a_1 = \log_\alpha \beta \pmod{p_1^{c_1}}, \ldots, a_k = \log_\alpha \beta \pmod{p_k^{c_k}}$. This can be done by examining all the possible values between 0 and p_1 or using another discrete logarithm algorithm such as Shanks' algorithm. Finally, we use the Chinese Remainder algorithm to determine the unique a. A detailed description of the algorithm is presented as Algorithm 6.3 in [33].

A straightforward implementation of the Pohlig-Hellman algorithm runs in time $O(c_i p_i)$ for each $\log_\alpha \beta \pmod{p_i^{c_i}}$. However, using Shanks' algorithm (which runs in time $O(\sqrt{p_i})$) to compute the smaller instances of the discrete log problem, we can reduce the overall running time to $O(c_i\sqrt{q_i})$. In our implementation, we use this strategy of using Pohlig-Hellman algorithm in conjunction with Shanks' algorithm for the runtime optimization.

In general, no efficient algorithm (i.e., polynomial time in $\log n$) is known for computing the discrete logarithm. In our setting, we will need to compute discrete logarithms in $\mathbb{Z}_p$. If $p - 1 = \prod p_i^{c_i}$, the discrete logs will cost $O(\sum c_i\sqrt{p_i})$ arithmetic operations in $\mathbb{Z}_p$. This will be intractable if $p-1$ has a large prime factor. (E. g., $p - 1 = 2q$, where q is a large prime.) We will choose p so that $p - 1$ has small prime factors, keeping the cost of computing the discrete logarithms low.

2.3 The Idea and an Example

In this section, we give a sequential description of our algorithm. Let $f = \sum_{i=1}^{t} c_i M_i$ be a polynomial, where $M_i = x_1^{e_{i1}} x_2^{e_{i2}} \cdots x_n^{e_{in}}$ are the t distinct monomials of f and $c_i \in \mathbb{Z}\backslash\{0\}$, with partial degrees $d_j = \deg_{x_j} f$, $1 \leq j \leq n$. Let $D_j \geq d_j$ denote the degree bounds for the respective variables and $T \geq t$ the term bound. For simplicity, we will assume $T = t$ as well as $D_j = d_j$ in this section. As in Ben-Or and Tiwari's algorithm, we proceed in two phases: The monomials M_i are determined in the first phase using probes to the black box and a linear generator based on those evaluations. Next, the coefficients c_i are determined in the second phase.

Let $q_1, \ldots, q_n$ be n pairwise relatively prime integers so that $q_i > D_i$ for $1 \leq i \leq n$ and $p = (\prod_{i=1}^{n} q_i) + 1$ is a prime. For a given set of $D_1, \ldots, D_n$, such a prime is relatively easy to construct: Let q_1 be the smallest odd number greater than D_1. For $2 \leq i \leq n-1$, choose q_i to be the smallest odd number greater than D_i such that $\gcd(q_i, q_j) = 1$ for $1 \leq j < i$. Now, let $a = q_1 \cdots q_{n-1}$. Then we need a prime of the form $p = a \cdot q_n + 1$. By Dirichlet's Prime Number Theorem, we know that there are infinitely many primes in the arithmetic progression $ab + 1$ [10]. So we just pick the smallest even number $q_n > D_n$ that is relatively prime to a and $a \cdot q_n + 1$ is prime.

Let ω be a primitive element of $\mathbb{Z}_p^*$, which can be found with a quick search: Choose a random $\omega \in \mathbb{Z}_p^*$ and compute $\omega^{(p-1)/p_i} \pmod{p}$ for each prime divisor p_i of $p-1$. If $\omega^{(p-1)/p_i} \not\equiv 1 \pmod{p}$ for each p_i, then ω is a primitive element. We already have the partial factorization $p - 1 = \prod_{i=1}^{n} q_i$ with pairwise relatively prime q_i, so finding the prime decomposition $p - 1 = \prod_{i=1}^{k} p_i^{e_i}$ is easy. Note that this is the method currently used by Maple's `primroot` routine.

Given $q_1, \ldots, q_n$, p, and ω as above, our algorithm starts by defining $\alpha_i = \omega^{(p-1)/q_i} \bmod p$ for $1 \leq i \leq n$. The α_i are primitive q_i-th roots of unity in $\mathbb{Z}_p$. We probe the black box to obtain $2T$ evaluations

$$v_i = f(\alpha_1^i, \alpha_2^i, \ldots, \alpha_n^i) \bmod p \text{ for } 0 \leq i \leq 2T - 1.$$

We then use these evaluations as the input for the Berlekamp-Massey algorithm [25] to obtain the linear generator $\Lambda(z)$ whose t roots are $m_i = M_i(\alpha_1, \alpha_2, \ldots, \alpha_n) \pmod{p}$, for $1 \leq i \leq t$. To find the roots of $\Lambda(z) \in \mathbb{Z}_p[z]$, we use a root finding algorithm such as Rabin's probabilistic algorithm presented in [29].

Now, take m_i for some $1 \leq i \leq t$ and note

$$m_i = M_i(\alpha_1, \ldots, \alpha_n) = \alpha_1^{e_{i1}} \times \cdots \times \alpha_n^{e_{in}} \pmod{p}.$$

Then we have

$$\log_\omega m_i = e_{i1} \log_\omega \alpha_1 + \cdots + e_{in} \log_\omega \alpha_n \pmod{(p-1)},$$

and since $\log_\omega \alpha_i = \frac{p-1}{q_i}$ from the construction $\alpha_i = \omega^{(p-1)/q_i}$, we see

$$\log_\omega m_i = e_{i1} \left(\frac{p-1}{q_1} \right) + \cdots + e_{in} \left(\frac{p-1}{q_n} \right) \pmod{(p-1)}.$$

Moreover, consider $\frac{p-1}{q_j} = \left(\prod_{k=1}^n q_k\right)/q_j = \prod_{k=1, k\neq j}^n q_k$ and recall q_i are pairwise coprime. Thus $q_j \mid \frac{p-1}{q_k}$ for any $1 \leq k \neq j \leq n$, i.e., $\frac{p-1}{q_k} \equiv 0 \pmod{q_j}$ for any $k \neq j$. Then it follows

$$\log_\omega m_i \equiv e_{ij} \left(\frac{p-1}{q_j} \right) \pmod{q_j}.$$

Now, since the q_i are relatively prime, $\frac{p-1}{q_j}$ is invertible modulo q_j, and we have

$$e_{ij} = \left(\frac{p-1}{q_j} \right)^{-1} \log_\omega m_i \pmod{q_j} \tag{2.3.1}$$

for $1 \leq i \leq t$, $1 \leq j \leq n$. Hence we obtain all monomials of f.

Here, we need to make an important observation. If $M_i(\alpha_1, \ldots, \alpha_n) = M_j(\alpha_1, \ldots, \alpha_n) \pmod{p}$ for some $1 \leq i \neq j \leq t$, we say the two monomials *collide*. If some monomials do collide, the rank of the linear system required to generate the linear generator $\Lambda(z)$ will be less than t, and in turn $\deg \Lambda(z) < t$. This creates a problem, because our algorithm depends on the degree of $\Lambda(z)$ to find the actual number of nonzero terms t. Fortunately, no monomial collision occurs in our algorithm, as we will show in Theorem 2.16. We will need the following lemma.

Lemma 2.15 ([26], Section 9.3.3)**.** *Let $q_1, \ldots, q_k$ be positive pairwise relatively prime integers, $Q = \prod_{i=1}^k q_i$, and $Q_i = Q/q_i$. Then the map*

$$\psi : \mathbb{Z}/Q\mathbb{Z} \longrightarrow \prod_{i=1}^k (\mathbb{Z}/q_i\mathbb{Z}) : x \longmapsto (x \bmod q_1, \ldots, x \bmod q_k)$$

is an isomorphism with

$$\psi^{-1}(a_1, \ldots, a_k) = (a_1 y_1 Q_1 + \cdots + a_k y_k Q_k) \bmod Q,$$

where $y_i Q_i \equiv 1 \bmod q_i$.

Theorem 2.16. *$M_i(\alpha_1, \ldots, \alpha_n) \neq M_j(\alpha_1, \ldots, \alpha_n) \pmod p$ for $1 \leq i \neq j \leq t$.*

Proof. Suppose $M_i(\alpha_1, \ldots, \alpha_n) = M_j(\alpha_1, \ldots, \alpha_n)$ for some $1 \leq i, j \leq t$. We will show $i = j$. Let $Q_k = \frac{p-1}{q_k} = \left(\prod_{h=1}^{n} q_h\right)/q_k$ for $1 \leq k \leq n$. We have

$$\begin{aligned} M_i(\alpha_1, \ldots, \alpha_n) &= \alpha_1^{e_{i1}} \times \cdots \times \alpha_n^{e_{in}} \\ &= (\omega^{(p-1)/q_1})^{e_{i1}} \times \cdots \times (\omega^{(p-1)/q_n})^{e_{in}} \\ &= \omega^{e_{i1}Q_1 + \cdots + e_{in}Q_n}. \end{aligned}$$

Hence $M_i(\alpha_1, \ldots, \alpha_n) = M_j(\alpha_1, \ldots, \alpha_n) \pmod p$ if and only if

$$\begin{aligned} &\omega^{e_{i1}Q_1 + \cdots + e_{in}Q_n} \equiv \omega^{e_{j1}Q_1 + \cdots + e_{jn}Q_n} \pmod p \\ \iff\ &e_{i1}Q_1 + \cdots + e_{in}Q_n \equiv e_{j1}Q_1 + \cdots + e_{jn}Q_n \pmod{(p-1)}. \end{aligned}$$

Let $a_{ik} = e_{ik}Q_k \pmod{q_k}$ and $y_k = Q_k^{-1} \pmod{q_k}$ for $1 \leq k \leq n$. (Note y_k exists for all k, because q_k are pairwise coprime and thus $\gcd(q_k, Q_k) = 1$.) Let ψ be the isomorphism from Lemma 2.15. Then

$$e_{i1}Q_1 + \cdots + e_{in}Q_n = a_{i1}y_1Q_1 + \cdots + a_{in}y_nQ_n = \psi^{-1}(a_{i1}, \ldots, a_{in}) \pmod{(p-1)},$$

and similarly,

$$e_{j1}Q_1 + \cdots + e_{jn}Q_n = a_{j1}y_1Q_1 + \cdots + a_{jn}y_nQ_n = \psi^{-1}(a_{j1}, \ldots, a_{jn}) \pmod{(p-1)}.$$

Thus

$$\psi^{-1}(a_{i1}, \ldots, a_{in}) = \psi^{-1}(a_{j1}, \ldots, a_{jn}).$$

However, ψ is an isomorphism, so

$$\psi^{-1}(a_{i1}, \ldots, a_{in}) = \psi^{-1}(a_{j1}, \ldots, a_{jn}) \iff (a_{i1}, \ldots, a_{in}) = (a_{j1}, \ldots, a_{jn}).$$

Moreover, $(a_{i1}, \ldots, a_{in}) = (a_{j1}, \ldots, a_{jn}) \iff a_{ik} = e_{ik}Q_k \equiv e_{jk}Q_k = a_{jk} \pmod{q_k}$ for all $1 \leq k \leq n$. Therefore

$$\begin{aligned} e_{ik}Q_k \equiv e_{jk}Q_k \pmod{q_k} &\iff e_{ik}Q_k - e_{jk}Q_k \equiv 0 \pmod{q_k} \\ &\iff (e_{ik} - e_{jk})Q_k \equiv 0 \pmod{q_k}. \end{aligned}$$

Again, $\gcd(q_k, Q_k) = 1$, so it follows that $e_{ik} - e_{jk} \equiv 0 \pmod{q_k}$, i.e., $q_k \mid (e_{ik} - e_{jk})$. Now, by the choice of q_k, we have $0 \leq e_{ik}, e_{jk} \leq D_k < q_k$ for $1 \leq k \leq n$. Hence $e_{ik} - e_{jk} = 0$. That is, $e_{ij} = e_{jk}$ for all $1 \leq k \leq n$ and $M_i(x_1, \ldots, x_n) = M_j(x_1, \ldots, x_n)$. Finally, recall that M_i are distinct monomials. Therefore, $i = j$, as claimed. □

Once we have found all the monomials of f, we proceed to determine the coefficients c_i by setting up the transposed Vandermonde system $A\bar{c} = \bar{v}$ described in (2.1.8) for Ben-Or and Tiwari's algorithm.

We demonstrate our algorithm with the following example.

Example 2.17. Let $f = 72xy^6z^5+37x^{13}+23x^3y^4z+87y^4z^3+29z^6+10 \in \mathbb{Z}[x, y, z]$. Suppose we are given a black box B that computes f. Here $n = 3, d_1 = \deg_x f = 13, d_2 = \deg_y f = 6, d_3 = \deg_z f = 6$, and $t = 6$. We will use the partial degree bounds $D_1 = 13, D_2 = 6$, and $D_3 = 6$ for the degrees d_x, d_y, and d_z, respectively as well as the term bound $T = t = 6$.

We pick three pairwise relatively prime numbers so that $q_i > D_i$ and $p = q_1q_2q_3 + 1$ is prime. Let our three numbers be $q_1 = 15, q_2 = 17$, and $q_3 = 14$. This determines our prime $p = 15 \cdot 17 \cdot 14 + 1 = 3571$. From now on, we proceed with all arithmetic operations modulo p. We then find ω, a random generator of $\mathbb{Z}_p^*$, using a simple search described earlier. Here, $\omega = 2$ is primitive modulo p. We compute

$$\alpha_1 = \omega^{\frac{p-1}{q_1}} = 1121, \alpha_2 = \omega^{\frac{p-1}{q_2}} = 1847, \text{ and } \alpha_3 = \omega^{\frac{p-1}{q_2}} = 2917.$$

Our evaluation points sent to B are

$$\beta_i = (\alpha_1^i, \alpha_2^i, \alpha_3^i) \text{ for } 0 \leq i \leq 2T - 1$$

as before. Let $v_i = f(\beta_i) \bmod p$. The $2T = 12$ evaluations are

$$\begin{array}{llll} v_0 = 258; & v_1 = 3079; & v_2 = 2438; & v_3 = 493; \\ v_4 = 3110; & v_5 = 2536; & v_6 = 336; & v_7 = 40; \\ v_8 = 2542; & v_9 = 2884; & v_{10} = 2882; & v_{11} = 201. \end{array}$$

Given these evaluations, we use the Berlekamp-Massey algorithm to compute the linear generator

$$\Lambda(z) = z^6 + 3554z^5 + 144z^4 + 3077z^3 + 2247z^2 + 3492z + 1769.$$

We compute the roots in $\mathbb{Z}_p$ using Rabin's algorithm to obtain

$$R = \{3191, 3337, 2913, 3554, 1305, 1\}.$$

Let $m_1 = M_1(\alpha_1, \alpha_2, \alpha_3) = 3191$ and compute $l_1 = \log_\omega m_1 = 2773$. Then by (2.3.1), the exponents of x, y and z in the first monomial M_1 are

$$\begin{aligned} e_{11} &= \left(\tfrac{q_1}{p-1}\right) l_1 \pmod{q_1} = \left(\tfrac{1}{17\cdot 14}\right)(2773) \pmod{15} = 1, \\ e_{12} &= \left(\tfrac{q_2}{p-1}\right) l_1 \pmod{q_2} = \left(\tfrac{1}{15\cdot 14}\right)(2773) \pmod{17} = 6, \\ e_{13} &= \left(\tfrac{q_3}{p-1}\right) l_1 \pmod{q_3} = \left(\tfrac{1}{15\cdot 17}\right)(2773) \pmod{14} = 5. \end{aligned}$$

We find $M_1 = xy^6z^5$. We repeat the process for the rest of the roots of $\Lambda(z)$ to find the monomials of f to be $xy^6z^5, x^{13}, x^3y^4z, y^4z^3, z^6$, and 1.

For the coefficients of f, we set up the transposed Vandermonde system of linear equations described in (2.1.8). That is, given $m_1 = 3191, m_2 = 3337, \ldots, m_6 = 1$, let

$$A = \begin{bmatrix} m_1^0 = 1 & m_2^0 = 1 & \cdots & m_6^0 = 1 \\ m_1^1 = 3191 & m_2^1 = 3337 & \cdots & m_6^1 = 1 \\ \vdots & \vdots & \ddots & \vdots \\ m_1^5 = 3157 & m_2^5 = 3571 & \cdots & m_6^5 = 1 \end{bmatrix} \quad \text{and} \quad \bar{v} = \begin{bmatrix} v_0 = 258 \\ v_1 = 3079 \\ \vdots \\ v_5 = 2536 \end{bmatrix}.$$

Solving the system $A\bar{c} = \bar{v}$ for $\bar{c} = [c_1, \ldots, c_6]^T$ finds the coefficients

$$c_1 = 72, c_2 = 37, c_3 = 23, c_4 = 87, c_5 = 29, \text{ and } c_6 = 10.$$

Hence we have completely determined our target polynomial

$$f(x, y, z) = 72xy^6z^5 + 37x^{13} + 23x^3y^4z + 87y^4z^3 + 29z^6 + 10,$$

which is the correct interpolation for the given B.

Remark 2.18. We chose $p = 3571$ in the above example, but in practice, we would choose p to be less than 2^{31} but as large as possible.

2.4 The Algorithm

Remark 2.19. Choosing inputs $q_1, q_2, \ldots, q_n$, and p can be achieved quickly by sequentially traversing through numbers from $D_i + 1$, as described in Section 2.3. As well, choosing the primitive element $\omega \in \mathbb{Z}_p^*$ can be done quickly using the method also described in Section 2.3. In our implementation, we use Maple's `primroot` routine to find ω.

Algorithm 2.1 Sparse Interpolation

Input: B: a *black box* B representing an unknown polynomial $f \in \mathbb{Z}[x_1, \ldots, x_n] \backslash \{0\}$
$(D_1, \ldots, D_n)$: partial degree bounds, $D_i \geq \deg_{x_i} f$
$(q_1, \ldots, q_n)$: n pairwise relatively prime integers, $q_i > D_i$ and $(\prod_{i=1}^{n} q_i) + 1$ prime
p: a prime number, $p = (\prod_{i=1}^{n} q_i) + 1$
ω: a primitive element modulo p
T: a bound on number of terms of f with nonzero coefficients, $T > 0$

Output: f_p, where $f_p \equiv f \pmod{p}$

1: Let $\alpha_i = \omega^{\frac{p-1}{q_i}}$ for $1 \leq i \leq n$.
2: Evaluate the black box B at $(\alpha_1^j, \alpha_2^j, \ldots, \alpha_n^j) \in \mathbb{Z}_p^n$ for $1 \leq j \leq 2T-1$.
Let $v_j = B(\alpha_1^j, \alpha_2^j, \ldots, \alpha_n^j) \bmod p$.
3: Apply the Berlekamp-Massey algorithm on the sequence v_j and obtain the linear generator $\Lambda(z)$. Set $t = \deg \Lambda(z)$.
4: Compute the set of t distinct roots $\{m_1, \ldots, m_t\} \in \mathbb{Z}_p^t$ of $\Lambda(z)$ modulo p using Rabin's algorithm.
5: **for** $i = 1 \rightarrow t$ **do**
6: Compute $l_i = \log_\omega m_i$ using the Pohlig-Hellman algorithm.
7: Let $e_{ij} = \left(\frac{q_j}{p-1}\right) l_i \pmod{q_j}$ for $1 \leq j \leq n$.
8: **end for**
9: Solve the linear system $S = \{c_1 m_1^i + c_2 m_2^i + \cdots + c_t m_t^i = v_i \mid 0 \leq i \leq t-1\}$ for $c_i \in \mathbb{Z}_p, 1 \leq i \leq t$. Here, $m_i = M_i(\alpha_1, \ldots, \alpha_n)$.
10: Define $f_p = \sum_{i=1}^{t} c_i M_i$, where $M_i = \prod_{j=1}^{n} x_j^{e_{ij}}$.
return f_p

Remark 2.20. Given that in all likely cases $p > 2$, we can assume the input p to be an odd prime without any significant consequence. In this case, $p - 1 = \prod_{i=1}^{n} q_i$ is even, so exactly one q_i will be even.

Remark 2.21. Our algorithm returns $f_p \equiv f \bmod p$ for the given prime p. To fully recover the integer coefficients of f, we can obtain more images of f modulo other primes and apply the Chinese Remainder algorithm to each of the t sets of coefficients. Assuming our initial choice of p does not divide any coefficient of f so that the number of terms in f_p is the same as the number of terms in f, we can generate the additional images without running the full algorithm again: Choose a new prime p^* and a random set of values $\alpha_1, \ldots, \alpha_n \in \mathbb{Z}_{p^*}$. Make t probes to the black box to obtain $v_j^* = B(\alpha_1^j, \ldots, \alpha_n^j) \bmod p^*$ for $0 \leq j \leq t-1$, where t is the number of terms in f_p. Compute $m_i^* = M_i(\alpha_1, \ldots, \alpha_n) \bmod p^*$ for $1 \leq i \leq t$, and solve the

transposed Vandermonde linear system $S^* = \{c_1^* m_1^i + c_2^* m_2^i + \cdots + c_t^* m_t^i = v_i^* \mid 0 \le i \le t-1\}$ as in Step 9 for $c_i^* \in \mathbb{Z}_{p^*}$, $1 \le i \le t$. This method of finding an additional image of f requires t evaluations and solving a $t \times t$ system.

If f_p has fewer terms than f because p does divide some nonzero coefficient of f, making an extra evaluation $v_t^* = B(\alpha_1^t, \ldots, \alpha_n^t) \bmod p^*$ and testing if v_t^* indeed equals $\sum_{i=1}^t c_i^* M_i(\alpha_1^t, \ldots, \alpha_n^t) \bmod p^*$ will detect any inconsistency caused by the missing monomial with high probability (Lemma 2.3). If an inconsistency is detected, we simply run the full algorithm again using another smooth prime p^*.

2.5 Complexity

In this section, we discuss the complexity of Algorithm 2.1. Let $d = \max\{d_i\}$. We will choose $p > (d+1)^n$ and count operations in $\mathbb{Z}_p$. Since $p > (d+1)^n$ an arithmetic operations in $\mathbb{Z}_p$ is not constant cost.

Theorem 2.22. *The expected total cost of our algorithm is*

$$O(TP(n,d,t) + nT + T^2 + t^2 \log p + t \sum_{i=1}^{n} \sqrt{q_i}) \text{ arithmetic operations in } \mathbb{Z}_p.$$

Proof. Step 1 does not contribute significantly to the overall cost of the algorithm.

For Step 2, the total cost of computing the evaluation points is $O((2T-1)n)$. Next, we need to count the cost of the probes to the black box. Let $P(n,d,t)$ denote the cost of one probe to the black box. Since the algorithm requires $2T$ probes, the total cost of the probes to the black box is $2TP(n,d,t)$. Hence the total cost of Step 2 is $O(2TP(n,d,t) + (2T-1)n) = O(TP(n,d,t) + nT)$.

In Step 3, the Berlekamp-Massey process as presented in [20] for $2T$ points costs $O(T^2)$ arithmetic operations modulo p using classical algorithm. It is possible to accelerate it to $O(M(T) \log T)$ using the Fast Euclidean algorithm, which we will discuss in Chapter 3. (See [11], Chapter 7).

In Step 4, in order to find the roots of $\Lambda(z)$ with $\deg \Lambda(z) = t$, we use Rabin's Las Vegas algorithm from [29], which we will review in more detail in Chapter 3. The algorithm tries to split $\Lambda(z)$ into linear factors by computing $g(z) = \gcd((z+\alpha)^{(p-1)/2} - 1, \Lambda(z))$ with randomly generated $\alpha \in \mathbb{Z}_p$. If we use the classical polynomial arithmetic, computing the power $(z+\alpha)^{(p-1)/2}$ for the initial GCD computation dominates the cost of the root-finding

algorithm. Thus if the implementations of polynomial multiplication, division and GCD use classical arithmetic, the cost of the step is $O(t^2 \log p)$ arithmetic operations modulo p.

In Step 6, we compute one discrete log $l_i = \log_\omega m_i$ using the Pohlig-Hellman algorithm which uses the known prime decomposition of $p-1$. We need a prime decomposition of $p-1$ before we can compute l_i. The factorization can be found quickly, since we already have the partial decomposition $p-1 = \prod_{i=1}^{n} q_i$. So this factorization does not significantly contribute to the overall cost of the step, especially since we only need to compute the factorization once for all l_i. Now, suppose

$$p-1 = \prod_{j=1}^{n} q_j = \prod_{j=1}^{n} \prod_{h=1}^{k_j} r_{jh}^{s_{jh}},$$

where r_{jh} are distinct primes, $s_{jh} > 0$, $k_j > 0$, and $q_j = \prod_{h=1}^{k_j} r_{jh}^{s_{jh}}$. The Pohlig-Hellman algorithm computes a series of smaller discrete logs $l_{jh} = \log_\omega m_i \pmod{r_{jh}^{s_{jh}}}$ and applies the Chinese Remainder algorithm to find l_i. Each of the smaller discrete logs costs $O(s_{jh}\sqrt{r_{jh}})$. Therefore, the cost of computing l_i is $O(\sum_{h=1}^{k_j} s_{jh}\sqrt{r_{jh}})$ plus the cost of the Chinese Remainder algorithm with $\sum_{j=1}^{n} k_j$ moduli. Note $r_{jh} \leq q_j$. If for some j and h, r_{jh} is large in relation to q_j, then s_{jh} is small, and it follows k_j must also be small. In this case, $O(\sum_{h=1}^{k_j} s_{jh}\sqrt{r_{jh}}) \in O(\sqrt{q_j})$. On the other hand, if q_j is smooth and r_{jh} are small, then $O(\sum_{h=1}^{k_j} s_{jh}\sqrt{r_{jh}})$ is close to $O(\log q_j)$. Hence, we have $O(\sum_{j,h} s_{jh}\sqrt{r_{jh}}) \in O(\sum_{j=1}^{n} \sqrt{q_j})$.

The cost of the Chinese Remainder theorem is $O(N^2)$, where N is the number of moduli, and we have $N = \sum_{j=1}^{n} k_j$, the number of distinct prime factors of $p-1$. There are at most $\log_2(p-1)$ factors of $p-1$, so the maximum cost of the Chinese remaindering step is $O((\log_2(p-1))^2)$. But we have $\lceil \log_2(p-1) \rceil = \lceil \log_2(\prod_{j=1}^{n} q_j) \rceil \leq \sum_{j=1}^{n} \lceil \log_2 q_j \rceil$, and $(\sum_{j=1}^{n} \log_2 q_j)^2 < \sum_{j=1}^{n} \sqrt(q_j)$ for large q_j. Thus the expected cost of Step 6 is $O(\sum_{j=1}^{n} \sqrt{q_j})$.

In Step 7 we multiply l_i by $\frac{q_j}{p-1}$ to obtain e_{ij} for $1 \leq j \leq n$. We can compute and store $\frac{q_j}{p-1}$ for $1 \leq j \leq n$ before the for-loop, which requires one inversion for $(p-1)^{-1}$ and n multiplications overall. These operations can be done in $O(n)$ time. As well, the n multiplications for $e_{ij} = \left(\frac{q_j}{p-1}\right) l_i \bmod q_i$ for $1 \leq j \leq n$ cost n multiplication in $\mathbb{Z}_p$ per iteration of the for-loop.

Adding the costs of Steps 6 and 7, we see that the total cost of the t iterations of the for-loop in Step 5 is $O(t(\sum_{j=1}^{n} \sqrt{q_j} + n))$. But $q_j > 1$ for all j and $\sum_{j=1}^{n} \sqrt{q_j} > n$, so the total cost is $O(t \sum_{j=1}^{n} \sqrt{q_j}) \in O(tn\sqrt{q})$, where $q \geq q_i$.

The transposed Vandermonde system of equations in Step 9 can be solved in $O(t^2)$ [34].

The expected total cost of our algorithm is therefore, as claimed,

$$O(TP(n,d,t) + nT + T^2 + t^2 \log p + t\sum_{i=1}^{n} \sqrt{q_i}).$$

□

Suppose $T \in O(t)$ and $q_i \in O(D_i)$ with $D_i \in O(d)$ for $1 \leq i \leq n$. (Remark 2.24 outlines how to find a term bound of $O(t)$.) Then $O(\log p) \in O(n \log d)$ from $\log(p-1) = \sum \log q_i$, and we can further simplify the cost of the algorithm to

$$O(tP(n,d,t) + nt^2 \log d + nt\sqrt{d}).$$

Given term bound T, Algorithm 2.1 makes exactly $2T$ probes. Therefore, if $T \in O(t)$, the algorithm makes $O(t)$ probes, which is a factor of nd smaller than the number of probes made in Zippel's algorithm and $O(n)$ smaller than that of Javadi and Monagan's. Moreover, the number of probes required is solely dependent on T. That is, Algorithm 2.1 is nonadaptive. Theorem 2.11 states that $2T$ is the fewest possible probes to the black box we can make while ensuring our output is correct given our nonadaptive approach. Therefore the algorithm is optimal in the number of evaluations it makes, minimizing one of the biggest bottlenecks in running time in interpolation algorithms.

Remark 2.23. The complexity of the algorithm shows that the black box evaluation, the root finding, and the discrete log steps dominate the running time. However, in practice, the discrete log step takes very little time at all. We will verify this claim later in Section 2.7, Benchmark #5.

Remark 2.24. Our algorithm requires a term bound $T \geq t$. However, it is often difficult in practice to find a good term bound for a given black box or be certain that an adequately large term bound was used. One way to solve this problem is to iterate Steps $1 \sim 3$ while increasing the term bound until the degree of the linear generator $\Lambda(z)$ in Step 3 is strictly less than the term bound. This strategy stems from the observation that $\deg \Lambda(z)$ is the rank of the system generated by $v_i = f(\alpha_1^i, \alpha_2^i, \ldots, \alpha_n^i) \bmod p$ for $0 \leq i \leq 2T-1$. In fact, this is exactly the linear system $V\bar{\lambda} = \bar{v}$ described in (2.1.6). By Theorem 2.6, if $T \leq t$ then $\text{rank}(V) = T$, so $\deg \Lambda(z) = T$ for $T \leq t$. That is, if we iterate until we get $\deg \Lambda(z) < T$ for some T, we can be sure the term bound is large enough and that we have found all t nonzero monomials.

To minimize redundant computation, the algorithm can be implemented to incorporate the previous evaluations, i.e., only compute the additional evaluation points $(\alpha_1^i, \ldots, \alpha_n^i)$ for $2T_{old} \leq i \leq 2T_{new} - 1$ and probe the black box. In this way, the algorithm makes exactly $2T$ probes to the black box in total, where T is the first tried term bound that gives $\deg \Lambda(z) < T$. If we use $T = 1, 2, 4, 8, 16, \ldots$, then we use at most double the number of probes necessary and $T \in O(t)$. For this method, the only significant additional cost is from generating the intermediate linear generators, which is $O(t^2 \log t)$. (Note $t \leq (d+1)^n$, so this $O(t^2 \log t)$ is overshadowed by the cost of the root finding step, $O(t^2 \log p)$, in the overall complexity.)

2.6 Optimizations

Let F be a field. Given a primitive N-th root of unity $\omega \in F$ and a univariate polynomial $f \in F[x]$ of degree at most $N-1$, the *discrete Fourier transform* (DFT) of f is the vector $[f(1), f(\omega), f(\omega^2), \ldots, f(\omega^{N-1})]$. The *fast Fourier transform* (FFT) efficiently computes the DFT in $O(N \log N)$ arithmetic operations in F. Due to its divide-and-conquer nature, the algorithm requires ω to be a 2^k-th root of unity for some k such that $2^k \geq N$. Thus, for $F = \mathbb{Z}_p$, we require $2^k \mid p-1$. If $p = 2^k r + 1$ is a prime for some $k, r \in \mathbb{N}$, r small, then we say p is a *Fourier prime*.

By Remark 2.20, exactly one q_i, namely q_n, is even for any given black box. If $q_n > D_n$ is chosen so that $q_n = 2^k > 2t$ for some $k \in \mathbb{N}$ then p is a Fourier prime, and we can use the FFT in our algorithm, particularly in computing $g(z) = \gcd((z+\alpha)^{(p-1)/2} - 1, \Lambda(z))$ for roots of $\Lambda(z)$ in Step 4.

Another approach to enable the use of the FFT in our algorithm is to convert the given multivariate polynomial into a univariate polynomial using the *Kronecker substitution* outlined in the following lemma.

Lemma 2.25 ([5], Lemma 1)**.** *Let K be an integral domain and $f \in K[x_1, \ldots, x_n]$ a polynomial of degree at most d. Then the substitution $x_i \mapsto X^{(d+1)^{i-1}}$ maps f to a univariate polynomial $g \in K[X]$ of degree at most $(d+1)^n$ such that any two distinct monomials M and M' in f map to distinct monomials in g.*

That is, given a multivariate f and partial degree bounds $D_i > \deg_{x_i} f$, $1 \leq i \leq n$, we can convert it to a univariate polynomial g by evaluating f at $(x, x^{D_1}, x^{D_1 D_2}, \ldots, x^{D_1 \cdots D_{n-1}})$

while keeping the monomials distinct. Then we need to find only a single integer q_1 so that $q_1 = 2^k r > \prod_{i=1}^{n} D_i$, k sufficiently large, and $p = q_1 + 1$ is a prime to use the FFT in our algorithm. Once the t nonzero terms of g are found, we can recover the monomials of f by inverting the mapping we used to convert f into g.

In Chapter 3, we introduce the *Fast Extended Euclidean Algorithm* (FEEA), which is a fast algorithm for finding the GCD of two polynomials. The polynomial $\Lambda(z)$ in Step 3 can be computed using FEEA to reduce the cost from $O(t^2)$ using the classical algorithm to $O(M(t)\log t)$ operations in $\mathbb{Z}_p$, where $M(t)$ is the cost of multiplying polynomials of degree at most t in $\mathbb{Z}_p[z]$ ([11], Chapter 7). If p is chosen to be a Fourier prime, then $\Lambda(z)$ can be computed in $O(t\log^2 t)$ operations.

The fast Fourier polynomial multiplication algorithm ([12], Algorithm 4.5) utilizes the FFT to compute the product of two polynomials of degrees m and n in $O((m+n)\log(m+n))$. The cost of $O(t^2 \log p)$ for computing the roots of $\Lambda(z)$ in Step 4 can be improved to $O(M(t)\log t(\log t + \log p))$ using fast multiplication and fast division (as presented in Algorithm 14.15 of [11]). Again, if p is chosen to be a Fourier prime, then $M(t) \in O(t\log t)$ and the roots of $\Lambda(z)$ can be computed in $O(t\log^2 t(\log t + n\log D))$ time. If $D \in O(d)$, then the total cost of Algorithm 2.1 is reduced to

$$O(tP(n,d,t) + t\log^3 t + nt\log^2 t\log d + nt\sqrt{d})).$$

However, $t \leq (d+1)^n$, so $\log t \in O(n\log d)$. Hence $t\log^3 t \in O(nt\log d\log^2)$.

2.7 Timings

In this section, we present the performance of our algorithm and compare it against Zippel's algorithm. The new algorithm uses a Maple interface that calls the C implementation of the algorithm. Some of the routines are based on Javadi's work for [16], which we optimized for speed. Zippel's algorithm also uses a Maple interface, which accesses the C implementation by Javadi. In addition, we include for the first three test sets the number of probes used for the Javadi and Monagan's algorithm as they appear in [16]. The rest of the test results for Javadi and Monagan's algorithm are not presented here due to testing environment differences. (In particular, the tests in [16] were run with a 31-bit prime $p = 2114977793$.)

Note that the black box model for the new algorithm is slightly different from that of the Zippel's. In both models, $B : \mathbb{Z}^n \mapsto \mathbb{Z}_p$ for a chosen p, but our algorithm requires a smooth

p that has relatively prime factors that are each greater than the given degree bounds. To address this difference, we first use our new algorithm to interpolate the underlying polynomial of a given black box modulo p of our choice and then proceed to interpolate the polynomial again with Zippel's algorithm with the same prime p.

We present the results of five sets of tests with randomly generated polynomials. We report the processing time and the number of evaluations made during the interpolation by each algorithm. All timings are in CPU seconds and were obtained using Maple's `time` routine for the overall times and the time.h package for C for individual routines. All tests were executed using Maple 15 on a 64 bit AMD Opteron 150 CPU 2.4 GHz with 2 GB memory running Linux.

We randomly generated for each test case a multivariate polynomial with coefficients in $\mathbb{Z}$ using Maple. The black box B takes in the evaluation point α as well as p and returns the polynomial evaluated at the given point of our choice, modulo p. In order to optimize computation time, the black box evaluation routine first computes all of x_j^i for $j = 1, \ldots, n$ and $i = 0, \ldots, d_j$, which takes $O(\sum_{i=1}^{n} d_i)$ arithmetic operations in $\mathbb{Z}_p$. Then the routine proceeds to compute each of the t terms of f by accessing the exponents of each variable and using values computed in the previous step before adding the t computed values and finally returns the values. This latter part of the routine can be done in $O(nt)$ arithmetic operations in $\mathbb{Z}_p$. Thus in our implementation $P(n, d, t)$, the cost of a single probe to the black box, is $O(nd + nt)$ arithmetic operations in $\mathbb{Z}_p$, where $d = \max\{d_i\}$.

Benchmark #1

In the first set of tests, we examine the impact the number of terms has on the computation time given a black box B for a polynomial f in $n = 3$ variables and of a relatively small degree $d = 30$. The multivariate polynomial for the i-th test polynomial is generated to have approximately $t = 2^i$ nonzero terms for $1 \leq i \leq 13$ using the following Maple command:

```
> f := randpoly( [x[1], x[2], x[3]], terms = 2^i, degree = 30);
```

For the i-th test, we use $D = 30$ and $T = 2^i$ as the degree and term bounds. The results are presented in Table 2.2. Our algorithm generally performs comparably to or better than Zippel's initially for $i \leq 10$ but becomes slower for $11 \leq i \leq 13$. This is due to the fact that although Zippel's algorithm does $O(nDt)$ probes for sparse polynomials, it does $O(t)$ for dense polynomials.

Table 2.2: Benchmark #1: $n = 3$, $d = 30 = D = 30$, $p = 34721$

i	t	T	New Algorithm		Zippel		Javadi
			Time	Probes ($2T$)	Time	Probes ($O(nDt)$)	Probes ($2nT+1$)
1	2	2	0.001	4	0.003	217	13
2	4	4	0.001	8	0.006	341	25
3	8	8	0.001	16	0.009	558	49
4	16	16	0.003	32	0.022	899	97
5	32	32	0.002	64	0.022	1518	193
6	64	64	0.007	128	0.060	2604	385
7	128	128	0.021	256	0.176	4599	769
8	253	256	0.066	512	0.411	6324	1519
9	512	512	0.229	1024	1.099	9672	3073
10	1015	1024	0.788	2048	2.334	12493	6091
11	2041	2048	2.817	4096	4.849	16182	12247
12	4081	4096	10.101	8192	9.127	16671	24487
13	5430	8192	21.777	16384	12.161	16927	32581

The data shows that the new algorithm makes fewer probes to the black box than both of the other two algorithms. As i increases, the respective polynomials become denser. In particular, for $i = 13$, the maximum possible number of terms is $t_{max} = \binom{n+d}{d} = \binom{33}{30} = 5456$. The bound $T = 2^{13} = 8192$ is greater than the actual number of monomials $t = 5430$. This inefficiency results in a significant increase in the processing time of the new algorithm. On the other hand, Zippel's algorithm's performance time does not increase as dramatically from $n = 12$ to $n = 13$. Given a completely dense polynomial, Zippel does $O(t)$ probes to interpolate it, and since $t \ll 2T$ in this case, Zippel's algorithm is more efficient. Given a sparse polynomial, Javadi and Monagan's algorithm makes $(2nT + 1) \in O(nT)$ probes, where the extra point is to check if the output is correct. Indeed, our algorithm consistently makes roughly a factor of $n = 3$ fewer evaluations for all but the last test cases.

To examine the impact of using a bad degree bound D to interpolate, we repeat the same set of tests with a degree bound $D = 100$. We present the results in Table 2.3. Both our algorithm and Javadi and Monagan's algorithm make the same number of probes as the first test, whereas the number of probes made by Zippel's algorithm increases roughly by the factor of 3, which is the increase in the degree bound, reflecting the fact Zippel makes $O(nDt)$ probes.

Table 2.3: Benchmark #1: $n = 3$, $d = 30$, $D = 100$ (bad bound), $p = 1061107$

i	t	T	New Algorithm		Zippel		Javadi
			Time	Probes ($2T$)	Time	Probes ($O(nDt)$)	Probes ($2nT+1$)
1	2	2	0.000	4	0.018	707	13
2	4	4	0.002	8	0.028	1111	25
3	8	8	0.001	16	0.050	1818	49
4	16	16	0.004	32	0.045	2929	97
5	32	32	0.003	64	0.085	4848	193
6	64	64	0.009	128	0.221	8484	385
7	128	128	0.028	256	0.602	14241	769
8	253	256	0.089	512	1.397	20604	1519
9	512	512	0.314	1024	3.669	31512	3073
10	1015	1024	1.112	2048	7.714	40703	6091
11	2041	2048	4.117	4096	15.944	49288	12247
12	4081	4096	15.197	8192	29.859	52722	24487
13	5430	8192	30.723	16384	38.730	53530	32581

The number of evaluations made by the new algorithm only depends on the term bound and therefore stays the same as in the first test. Nevertheless, the overall timings of the new algorithm increases with the bad degree bound. This is largely due to the root finding step with cost $O(t^2 \log p) \in O(t^2 n \log D)$ arithmetic operations in $\mathbb{Z}_p$. The cost of the discrete log step is $O(n\sqrt{D})$ arithmetic operations in $\mathbb{Z}_p$ and also increases with D. However, even the increased cost does not impact the overall timing significantly. In Benchmark #5, we present the breakdown of the timings and verify these explanations.

Benchmark #2

The second benchmarking set uses polynomials in $n = 3$ variables with approximately 2^i nonzero terms for the i-th polynomial. All polynomials will be of total degree approximately 100, which is larger than the 30 in the first set. This time, the polynomials are much more sparse than those for the first set. We will use degree bound $D = 100$. We generate the test cases using the following Maple code for $1 \leq i \leq 13$.

```
> f := randpoly( [x[1], x[2], x[3]], terms =  2^i, degree = 100);
```

Table 2.4 shows the result of the tests. The number of probes for our new algorithm stays the same as in the first set of tests even though the total degree of the polynomials

Table 2.4: Benchmark #2: $n = 3$, $d = D = 100$, $p = 1061107$

i	t	T	New Algorithm		Zippel		Javadi
			Time	Probes ($2T$)	Time	Probes ($O(nDt)$)	Probes ($2nT+1$)
1	2	2	0.000	4	0.016	505	-
2	4	4	0.001	8	0.020	1111	-
3	8	8	0.000	16	0.031	1919	49
4	16	16	0.001	32	0.074	3535	97
5	31	32	0.004	64	0.145	5858	187
6	64	64	0.010	128	0.350	10807	385
7	127	128	0.029	256	0.940	18988	763
8	254	256	0.092	512	2.726	32469	1519
9	511	512	0.319	1024	8.721	57242	3067
10	1017	1024	1.129	2048	28.186	98778	6103
11	2037	2048	4.117	4096	87.397	166751	12223
12	4076	4096	15.394	8192	246.212	262802	24457
13	8147	8192	56.552	16384	573.521	363226	48883

are higher for this second set.

Benchmark #3

The third benchmarking set uses polynomials in $n = 6$ variables with total degree and degree bound of 30. The i-th polynomial has roughly 2^i nonzero terms. The problem set is generated with the following Maple code for $1 \leq i \leq 13$. Since the maximum possible number of terms is $\binom{6+30}{30} = 1947792 \gg 8192 = 2^{13}$, the test polynomials are all sparse.

```
> f := randpoly([x[1],x[2],x[3],x[4],x[5],x[6]], terms=2^i, degree=30);
```

We present the results of the timings in Table 2.5. While the overall interpolation times increase in comparison to the second benchmarking set, the new algorithm still makes the same number of probes to the black box. The time increases we see are again mostly due to the increased cost of the root finding step.

Benchmark #4

To better highlight the strength of our new algorithm, we test with this set of extremely sparse polynomials wherein we hold the number of terms of f constant and increase the

Table 2.5: Benchmark #3: $n = 6$, $d = D = 30$, $p = 2019974881$

i	t	T	New Algorithm		Zippel		Javadi
			Time	Probes ($2T$)	Time	Probes ($O(nDt)$)	Probes ($2nT + 1$)
1	2	2	0.001	4	0.024	465	-
2	3	4	0.001	8	0.016	744	-
3	8	8	0.001	16	0.026	1333	97
4	16	16	0.002	32	0.045	2418	193
5	31	32	0.004	64	0.102	4340	373
6	64	64	0.014	128	0.298	8339	769
7	127	128	0.041	256	0.868	14570	1525
8	255	256	0.171	512	3.019	27652	3061
9	511	512	0.476	1024	10.499	50592	6133
10	1016	1024	1.715	2048	36.423	91171	12193
11	2037	2048	6.478	4096	133.004	168299	24445
12	4083	4096	24.733	8192	469.569	301103	48997
13	8151	8192	95.066	16384	1644.719	532673	97813

degrees of the polynomials. The fourth benchmarking set uses polynomials in $n = 3$ variables with the number of nonzero terms t around 100. The i-th polynomial is roughly of degree 2^i. We use the following Maple code to generate the polynomials.

```
> f := randpoly( [x[1], x[2], x[3]], terms = 100, degree = 2^i);
```

Table 2.6: Benchmark #4: $n = 3$, $T = 100$

i	d	p	New Algorithm		Zippel	
			Time	Probes	Time	Probes
1	2	61	0.002	200	0.000	36
2	4	211	0.007	200	0.003	121
3	8	991	0.008	200	0.019	450
4	16	8779	0.012	200	0.040	1532
5	32	43891	0.014	200	0.121	3762
6	64	304981	0.016	200	0.344	9330
7	128	2196871	0.021	200	0.954	20382
8	256	17306381	0.024	200	2.851	45746
9	512	137387581	0.031	200	8.499	94392
10	1024	1080044551	0.042	200	30.661	200900

Table 2.6 shows the results of the tests. The new algorithm makes 200 probes for each of the ten test cases, whereas Zippel's algorithm makes more than 200,000 probes for $i = 10$.

As a result, the new algorithm is significantly faster than Zippel's.

Benchmark #5

In this section, we present the cost of the key steps of our algorithm for two sets of tests using the polynomials we used for Benchmark #2 (Table 2.4) and at the same time demonstrate in detail the effect of increasing the degree bound.

Table 2.7: Benchmark #5: $n = 3$, $d = 100$, $p = 1061107 = 101 \cdot 103 \cdot 102 + 1$

i	T	Total Time	Black Box Probes	Berlekamp-Massey	Root Finding	Discrete Logs	Linear Solve
1	2	0.000	0.00	0.00	0.00	0.00	0.00
2	4	0.001	0.00	0.00	0.00	0.00	0.00
3	8	0.000	0.00	0.00	0.00	0.00	0.00
4	16	0.001	0.00	0.00	0.00	0.00	0.00
5	32	0.004	0.00	0.00	0.00	0.00	0.00
6	64	0.010	0.00	0.00	0.00	0.01	0.00
7	128	0.029	0.01	0.00	0.00	0.01	0.00
8	256	0.092	0.02	0.01	0.05	0.01	0.00
9	512	0.361	0.07	0.01	0.29	0.01	0.02
10	1024	1.129	0.24	0.07	0.70	0.02	0.08
11	2048	4.117	0.94	0.27	2.52	0.04	0.30
12	4096	15.394	3.75	1.09	9.04	0.07	1.19
13	8192	56.552	14.84	4.49	32.01	0.14	4.73

Table 2.8: Benchmark #5: $n = 3$, $d = 100$, $p = 1008019013 = 1001 \cdot 1003 \cdot 1004 + 1$

i	T	Total Time	Black Box Probes	Berlekamp-Massey	Root Finding	Discrete Logs	Linear Solve
1	2	0.000	0.00	0.00	0.00	0.00	0.00
2	4	0.001	0.00	0.00	0.00	0.00	0.00
3	8	0.002	0.00	0.00	0.00	0.01	0.00
4	16	0.004	0.00	0.00	0.00	0.00	0.00
5	32	0.010	0.00	0.00	0.00	0.01	0.00
6	64	0.012	0.00	0.00	0.01	0.00	0.00
7	128	0.036	0.01	0.00	0.02	0.01	0.00
8	256	0.114	0.02	0.00	0.08	0.01	0.01
9	512	0.397	0.07	0.02	0.37	0.01	0.02
10	1024	1.428	0.25	0.06	1.00	0.04	0.06
11	2048	5.337	0.95	0.23	3.81	0.07	0.23
12	4096	20.413	3.77	0.97	14.39	0.14	0.90
13	8192	77.481	14.83	3.96	54.44	0.28	3.63

Table 2.7 presents the breakdown of the timings from Benchmark #2, where $q_1 = 101, q_2 = 103, q_3 = 102$, and $p = 1061107$ were used. Table 2.8 shows the result of the running the tests with a larger 31-bit prime $p = 1008019013$ with $q_1 = 1001, q_2 = 1003$, and $q_3 = 1004$. Note that the latter setup is equivalent to using a bad degree bound $D = 1000$ for the degree 100 polynomials.

The data shows that for $i \geq 10$, the cost of the Berlekamp-Massey process, root-finding, and linear solve steps all grow roughly by a factor of 4 as T doubles, showing a quadratic increase. The breakdowns also verify our earlier statement for Benchmark #1 that the increase in the runtime when using a bad degree bound D is caused by the increase in the cost of the root-finding and discrete log steps. Moreover, the cost of root finding quickly becomes the biggest part of the total time as t increases in both tests. In contrast, the cost of the discrete log step remains very small compared to the overall cost, so the growth in the discrete log step does not contribute significantly to the overall cost of the interpolation.

Chapter 3

Fast Polynomial GCD

In this chapter, we are interested in the problem of computing polynomial GCDs over finite fields. In particular, we will work over $\mathbb{Z}_p$, the field of integers modulo p. First, we consider an example illustrating the importance of fast GCD computation. We then present the classical Euclid's algorithm for computing the GCD of two polynomials, followed by a fast variation of the algorithm, known as the Fast Extended Euclidean algorithm (FEEA).

The idea for the fast GCD algorithm was proposed by Lehmer in [24] for integer GCD computation. For integers of length n, Knuth [23] proposed a version of the fast algorithm with $O(n \log^5 n \log\log n)$ time complexity in 1970, which was improved by Schönhage [30] to $O(n \log^2 n \log\log n)$ in 1971. In 1973, Moenck [27] adapted Schönhage's algorithm to work with polynomials of degree n in $O(n \log^{a+1} n)$ time complexity, assuming fast multiplication in time complexity $O(n \log^a n)$ and division at least log reducible to multiplication. We develop the fast Euclidean algorithm for polynomials, as presented in von zur Gathen and Gerhard [11], which runs in $O(M(n) \log n)$ time complexity, where $M(n)$ is the cost of multiplying two polynomials of degree at most n. We have implemented the traditional and the fast algorithms for polynomials and present in Section 3.5 a comparison of their performance.

As shown in Chapter 2, our sparse interpolation algorithm requires an efficient root-finding algorithm for univariate polynomials in $\mathbb{Z}_p[x]$. We use Rabin's probabilistic algorithm, presented in Rabin's 1980 paper [29]. It finds roots of a univariate polynomial over $\mathbb{F}_q$ by computing a series of GCDs. We describe the algorithm here and show that the cost of computing the GCDs has a large impact on the cost of identifying the roots of a polynomial.

3.0.1 Rabin's Root Finding Algorithm

Let $\mathbb{F}_q$ be a fixed finite field, where $q = p^n$ for some odd prime p and $n \geq 1$. Suppose we are given a polynomial $f \in \mathbb{F}_q[x]$ with $\deg f = d > 0$ and want to find all $\alpha \in \mathbb{F}_q$ such that $f(\alpha) = 0$. We will need the following lemma.

Lemma 3.1. *In* $\mathbb{F}_q$, $x^q - x = \prod_{\alpha \in \mathbb{F}_q}(x - \alpha)$.

Proof. Recall that in a finite field with q elements, $\mathbb{F}_q^*$ is a multiplicative group of order $q - 1$. So $a^{q-1} = 1$ for any $a \in \mathbb{F}_q^*$. Then we have $a^q = a$, i.e., $a^q - a = 0$ for all $a \in \mathbb{F}_q^*$. Moreover, $0^q - 0 = 0$. Thus any $a \in \mathbb{F}_q$ is a solution to the equation $x^q - x = 0$. That is, $(x - a) \mid (x^q - x)$ for all $a \in \mathbb{F}_q$. Therefore we see $x^q - x = \prod_{\alpha \in \mathbb{F}_q}(x - \alpha)$, as claimed. □

Rabin's algorithm first computes $f_1 = \gcd(f, x^q - x)$. Let $k = \deg f_1$. By Lemma 3.1, f_1 is the product of all distinct linear factors of f in $\mathbb{F}_q[x]$, so we can write

$$f_1(x) = (x - \alpha_1) \cdots (x - \alpha_k), \quad k \leq d,$$

where $\alpha_1, \alpha_2, \ldots, \alpha_k \in \mathbb{F}_q$ are all distinct roots of f. Next, the algorithm exploits the factorization

$$x^q - x = x(x^{(q-1)/2} - 1)(x^{(q-1)/2} + 1)$$

to further separate the linear factors. Let $f_2 = \gcd(f_1, x^{(q-1)/2} - 1)$. Then all the α_i satisfying $\alpha_i^{(q-1)/2} - 1 = 0$ will also satisfy $(x - \alpha_i) | f_2$ while the rest of the α_i satisfy $\alpha_i^{(q-1)/2} + 1 = 0$ or $\alpha_i = 0$ instead and thus $(x - \alpha_i) \nmid f_2$.

A problem arises at this point, because there is no guarantee that $f_2 \neq 1$ or $f_2 \neq f_1$, in which case we have no new information. As a solution, Rabin introduces randomization to try to split f_1: after computing f_1, Rabin's algorithm randomly chooses $\delta \in \mathbb{F}_q$ and computes $f_\delta = \gcd(f_1, (x + \delta)^{(q-1)/2} - 1)$. This step is motivated by the observation that $0 < \deg f_\delta < k = \deg f_1$ with high probability.

Example 3.2. The probability of getting $0 < \deg f_\delta < k$ can be shown to be at least $1 - \left(\frac{q-1}{2q}\right)^k - \left(\frac{q+1}{2q}\right)^k$. For $k \geq 2$, this probability is minimized when $q = 4$ and $k = 2$, giving the lower bound of $\frac{4}{9}$. (Refer to the proof of Theorem 8.11 in [12] for details.)

Remark 3.3. This algorithm as presented in [29] uses $f_1 = \gcd(x^{q-1} - 1, f)$. In it, Rabin showed the probability $0 < \deg f_\delta < \deg f_1$ is at least $\frac{q-1}{2q} \approx \frac{1}{2}$ and conjectured that this probability is at least $1 - (\frac{1}{2})^{k-1} + O(\frac{1}{\sqrt{q}})$, which was proven by Ben-Or in [2].

If we are successful (i.e., we have that $0 < \deg f_\delta < \deg f_1$), then the algorithm proceeds to recursively compute the roots of f_δ and f_1/f_δ. On the other hand, if $f_\delta = 1$ or $f_\delta = f_1$, then the algorithm chooses another δ at random and tries again to split f_1. We repeat this process of choosing a random δ and computing f_δ until we find a satisfactory f_δ that splits f_1. We formally describe Rabin's root finding algorithm in Algorithm 3.1.

Algorithm 3.1 FIND_ROOTS(f, q)

– *Main routine*

Input: $f \in \mathbb{F}_q[x]$, $d = \deg f > 0$
Output: Distinct roots of $f = 0$ in $\mathbb{F}_q$

1: $f_1 \longleftarrow \gcd(f, x^q - x)$ /* monic GCD */
2: **return** ROOTS(f_1, q)

– *Subroutine:* ROOTS(f_1, q)

1: **if** $f_1 = 1$ **then return** $\{\}$
2: **if** $\deg f_1 = 1$, i.e., $f_1 = x - \alpha$ **then return** $\{\alpha\}$
3: $f_2 \longleftarrow 1$
4: **while** $f_2 = 1$ or $f_2 = f_1$ **do**
5: choose at random $\delta \in \mathbb{F}_q$
6: $f_2 \longleftarrow \gcd(f_1, (x - \delta)^{(q-1)/2} - 1)$ /* monic GCD */
7: **end while**
8: **return** ROOTS(f_2, q) $\cup$ ROOTS(f_1/f_2, q)

Remark 3.4. We can optimize the computation for $\gcd(f_1, (x-\delta)^{(q-1)/2}-1)$ in Step 6 of the subroutine ROOTS. First, $\gcd(f_1, (x-\delta)^{(q-1)/2}-1) = \gcd(f_1, (x-\delta)^{(q-1)/2}-1 \bmod f_1)$, so it is sufficient to compute $(x-\delta)^{(q-1)/2}-1 \bmod f_1$. That is, instead of working with a degree $\frac{q-1}{2}$ polynomial $(x-\delta)^{(q-1)/2}-1$, we can reduce it to $(x-\delta)^{(q-1)/2}-1 \bmod f_1$ so that the GCD computation involves two polynomials of degrees at most k, since $\deg((x-\delta)^{(q-1)/2} - 1 \bmod f_1) < k$. Moreover, rather than naively computing $(x-\delta)^{(q-1)/2}$ by multiplying $(x-\delta)$ by itself $\frac{q-1}{2} - 1$ times, modulo f_1, we can use the technique known as square-and-multiply and achieve the exponentiation in $O(\log q)$ multiplications and divisions of polynomials of degrees at most $2k \leq 2d$. (See [11], Algorithm 4.8.) Using classical polynomial arithmetic, this costs $O(d^2 \log q)$ arithmetic operations in $\mathbb{F}_q$. Note that we can optimize Step 1 of the main algorithm FIND_ROOTS in the same way.

One can show that the expected total number of arithmetic operations made by Algorithm 3.1 is $O(d^2 \log d \log q)$ for the powering plus $O(d^2 \log d)$ for computing the GCDs.

Gerhard and von zur Gathen [11] showed that if fast polynomial multiplication is used this can be reduced to $O(M(d)\log q \log d + M(d)\log^2 d)$ arithmetic operations in $\mathbb{F}_q$.

Remark 3.5. In case of $p = 2$, the algorithm described above is not directly applicable. However, by making a modification to the algorithm so that it uses the trace polynomial $Tr(x) = x + x^2 + \cdots + x^{2^{n-1}}$ in place of $x^q - x$, we can apply the same overall strategy to find roots of f in $\mathbb{F}_q$.

Remark 3.6. Notice that there is a natural relationship between the problem of finding roots and the problem of factoring polynomial. In fact, root finding can be used to find polynomial factorization, as shown in [4] and [29]. For example, given the problem of factoring a polynomial $f \in \mathbb{Z}_p[x]$ into irreducible factors, Rabin reduces it to the problem of finding roots of the same polynomial. Thus we see that polynomial factorization is, in turn, another example of an application of polynomial GCD.

3.1 Preliminaries

Here, we introduce some notation. Throughout this section, let D denote an integral domain. The definitions and algorithms presented here are adopted from von zur Gathen and Gerhard [11] and Geddes *et al.* [12].

Definition 3.7. An element $u \in D$ is called a *unit* if there is a multiplicative inverse of u in D, i.e., there is $v \in D$ such that $uv = vu = 1$.

Definition 3.8. Two elements $a, b \in D$ are *associates* if $a \mid b$ and $b \mid a$, which is denoted by $a \sim b$.

In an integral domain D, if two elements a and b are associates, then $a = ub$ for some unit $u \in D$. By using the fact that $\sim$ is an equivalence relation on D, we can partition D into *associate classes* $[a] = \{b : b \sim a\}$ so that each class is formed by selecting a thus far unchosen element of D and collecting the set of all associates of D. A single element from each class is chosen as the canonical representative and is defined to be *unit normal*. For example, in $\mathbb{Z}$, the associate classes are $\{0\}, \{-1, 1\}, \{-2, 2\}, \ldots$, and the nonnegative element from each class is defined to be unit normal. If D is a field, all nonzero elements are associates of each other, and the only unit normal elements are 0 and 1.

Definition 3.9. Let $\mathrm{n}(a)$ denote the *normal part* of $a \in D$, the unit normal representative of the associate class containing a. For nonzero a, the *unit part* of a is the unique unit $\mathrm{u}(a) \in D$ such that $a = \mathrm{u}(a)\,\mathrm{n}(a)$. If $a = 0$, denote $\mathrm{n}(0) = 0$ and $\mathrm{u}(0) = 1$.

Definition 3.10. Let $a, b \in D$. Then $c \in D$ is a *greatest common divisor* (GCD) of a and b if

(i) $c \mid a$ and $c \mid b$,

(ii) $c \mid r$ for any $r \in D$ that divides both a and b.

Given $a, b \in D$, if $c, d \in D$ are both GCDs of a and b then it follows that $c \sim d$. As well, if c is a GCD of a and b, then any associate of c is also a GCD of a and b. Therefore, it is important to establish that the notation $g = \gcd(a, b)$ refers to the unique unit normal GCD g of a and b whenever the unit normal elements for D are defined. If D is a field, then $\gcd(a, b) = 0$ for $a = b = 0$ and $\gcd(a, b) = 1$ otherwise.

Remark 3.11. For any $a \in D$, $\gcd(a, 0) = n(a)$.

Definition 3.12. A nonzero element p in D is a prime if p is not a unit and whenever $p = ab$ for some $a, b \in D$, either a or b is a unit.

Definition 3.13. An integral domain D is a *unique factorization domain* (UFD) if for all nonzero a in D either a is a unit or a can be expressed as a finite product of primes such that this factorization into primes is unique up to associates and reordering.

Definition 3.14. A *Euclidean domain* E is an integral domain with an associated *valuation* function $\mathrm{v} : E\backslash\{0\} \to \mathbb{Z}_{\geq 0}$ with the following properties:

1. For all $a, b \in E\backslash\{0\}, \mathrm{v}(ab) \geq \mathrm{v}(a)$;
2. For all $a, b \in E$ with $b \neq 0$, there exist elements $q, r \in D$ such that $a = bq + r$ where r satisfies either $r = 0$ or $\mathrm{v}(r) < \mathrm{v}(b)$.

Example 3.15. The integers $\mathbb{Z}$ and the polynomial ring $F[x]$, where F is a field, are Euclidean domains, with valuations $\mathrm{v}(a) = |a|$ and $\mathrm{v}(a) = \deg a$ respectively.

3.2 The Euclidean Algorithm

Given the problem of computing the GCD of two given elements of a Euclidean domain E, the Euclidean algorithm computes a series of divisions and computing the GCD of two smaller elements which are remainders from the divisions in E. The correctness of the algorithm follows from Theorem 3.16.

Theorem 3.16 ([12], Theorem 2.3)**.** *Let E be a Euclidean domain with a valuation v. Let $a, b \in D$, where $b \neq 0$. Suppose some quotient $q \in E$ and remainder $r \in E$ satisfy*

$$a = bq + r \text{ with } r = 0 \text{ or } \mathrm{v}(r) < \mathrm{v}(b).$$

Then $\gcd(a, b) = \gcd(b, r)$.

Definition 3.17. Given a division $a \div b$, let a rem b and a quo b denote the remainder r and the quotient q of the division, so that r and q satisfy $a = bq + r$ with $r = 0$ or $\mathrm{v}(r) < \mathrm{v}(b)$.

We will illustrate with an example the Euclidean algorithm for polynomials in $F[x]$, where F is a field and $\mathrm{v}(a) = \deg(a)$. However, before we discuss the Euclidean algorithm for polynomials, we need to choose which elements are the associate class representatives for $F[x]$. Recall that in a field F, all nonzero elements are units and thus form one associate class with 1 as the class representative. Then for a polynomial $f \in F[x]$, $f \sim af$ for any $a \in F\backslash\{0\}$, and the associate classes of $F[x]$ are formed by nonzero scalar multiples of the polynomials of $F[x]$. Hence, a reasonable choice for the representative of an associate class in $F[x]$ is the *monic* polynomial, i.e., the polynomial whose leading coefficient is 1. That is, if g is a GCD of some $a, b \in F[x]$, $\gcd(a, b) = g/\operatorname{lc}(g)$, where $\operatorname{lc}(g)$ denotes the leading coefficient of g. Note the leading coefficient of $g \in F[x]$ satisfies the definition of the unit part for g. Thus we can use $\mathrm{u}(f) = \operatorname{lc}(f)$ and $\mathrm{n}(f) = f/\operatorname{lc}(f)$ for any $f \in F[x]$. As usual, if $f = 0$ then $\mathrm{u}(f) = 1$ and $\mathrm{n}(f) = 0$.

Example 3.18 (Euclidean Algorithm for Polynomials)**.** Let $p = 17$ and $E = \mathbb{Z}_p[x]$. Since 17 is a prime, $\mathbb{Z}_p$ is a field, and therefore E is a Euclidean domain. Suppose we are given $a(x) = 8x^3 + 3x^2 - 2x - 3$ and $b(x) = 3x^3 - 6x^2 + 6x - 8$. To compute $\gcd(a(x), b(x))$, we

proceed as follows.

$$\begin{aligned} a(x) &= -3 \cdot b(x) + (2x^2 - x + 7) \pmod{17} \\ &\Rightarrow \gcd(a(x), b(x)) = \gcd(b(x), 2x^2 - x + 7) \\ b(x) &= (-7x + 2) \cdot (2x^2 - x + 7) + (6x - 5) \pmod{17} \\ &\Rightarrow \gcd(b(x), 2x^2 - x + 7) = \gcd(2x^2 - x + 7, 6x - 5) \\ 2x^2 - x + 7 &= (6x + 2) \cdot (6x - 5) + 0 \\ &\Rightarrow \gcd(2x^2 - x + 7, 6x - 5) = \gcd(6x - 5, 0) \end{aligned}$$

Thus $\gcd(a(x), b(x)) = \gcd(6x-5, 0)$, and $\gcd(6x-5) = n(6x-5)$ by Remark 3.11. Finally, $n(6x - 5) = x + 2$, so $\gcd(a(x), b(x)) = x + 2$ in $\mathbb{Z}_{17}[x]$.

The Euclidean algorithm we used in our example can be formally described as follows.

Algorithm 3.2 Euclidean Algorithm

Input: $a, b \in E$, where E is a Euclidean domain with valuation v
Output: $\gcd(a, b)$

```
1: r_0 ⟵ a; r_1 ⟵ b
2: i = 1
3: while r_i ≠ 0 do
4:     r_{i+1} ⟵ r_{i-1} rem r_i
5:     i ⟵ i + 1
6: end while
7: g ⟵ n(r_{i-1})
8: return g
```

Definition 3.19. Let $a, b \in E$. The sequence $r_0, r_1, \ldots, r_l, r_{l+1}$ with $r_{l+1} = 0$ obtained from computing $\gcd(a, b)$ using the Euclidean algorithm is called the *Euclidean remainder sequence*.

3.2.1 Complexity of the Euclidean Algorithm for $F[x]$

We now examine the complexity of the Euclidean algorithm. We will focus on the case $E = F[x]$ for some field F, for which the degree function is the valuation. We will assume classical division and count the number of arithmetic operations in F.

Suppose we are given $f, g \in F[x]$ with $\deg f = n \geq \deg g = m \geq 0$. Let l be the number of iterations of the while-loop. The cost of the algorithm is the cost of the divisions

executed in the while-loop plus the cost of computing $\mathrm{n}(r_l)$. Let $d_i = \deg r_i$ for $0 \le i \le l$ and $d_{l+1} = -\infty$. From $r_{i+1} = r_{i-1} \text{ rem } r_i$, we have $d_{i+1} < d_i$ or $d_{i+1} \le d_i - 1$ for $2 \le i < l$. The number of iterations of the while-loop in the algorithm is therefore bounded by $\deg g + 1 = m + 1$.

We now find the cost of a single division. Let $q_i = r_{i-1} \text{ quo } r_i$. We have the degree sequence $d_0 = n \ge d_1 = m > d_2 > \cdots > d_l$, and it follows $\deg q_i = d_{i-1} - d_i$. Recall that given $a, b \in F[x]$, where $\deg a \ge \deg b \ge 0$, the division with remainder $a \div b$ costs, counting subtractions as additions, at most $(2 \deg b + 1)(\deg(a \text{ quo } b) + 1)$ additions and multiplications in F plus one division for inverting $\mathrm{lc}(b)$ ([11], Chapter 2). Thus, dividing r_{i-1}, a polynomial of degree d_{i-1}, by r_i, a polynomial of degree $d_i < d_{i-1}$, requires at most $(2d_i + 1)((d_{i-1} - d_i) + 1)$ additions and multiplications plus one inversion in F. Then, combining the number of iterations and the main cost of each loop, we see the total cost of the while-loop portion of the Euclidean algorithm is

$$\sum_{1 \le i \le l} (2d_i + 1)(d_{i-1} - d_i + 1) \tag{3.2.1}$$

additions and multiplications plus l inversions in F.

It can be shown that given a degree sequence $n = d_0 \ge m = d_1 > d_2 > \cdots > d_l \ge 0$, the maximum value for the sum in (3.2.1) occurs when the sequence is *normal*, i.e., when $d_i = d_{i-1} - 1$ for $2 \le i \le l$. Note in this case $l = m + 1$, which is the maximum possible number of divisions. Moreover, it can also be shown that for random inputs, it is reasonable to assume the degree sequence to be normal, except in the first division step where it is possible $n - m \gg 1$. So, we consider the worst case $d_i = m - i + 1$ for $2 \le i \le l = m + 1$ to obtain a bound for the maximal number of arithmetic operations performed in the while-loop. Then the expression in (3.2.1) can be simplified to

$$\begin{aligned} &(2m+1)(n-m+1) + \sum_{2 \le i \le m+1} [2(m-i+1)+1] \cdot 2 \\ &\quad = (2m+1)(n-m+1) + 2(m^2 - m) + 2m \\ &\quad = 2nm + n + m + 1. \end{aligned} \tag{3.2.2}$$

To compute $g = \mathrm{n}(r_l)$, we find the leading coefficient $\mathrm{lc}(r_l)$ and then multiply each of the terms of r_l by the inverse of $\mathrm{lc}(r_l)$. This process consists of at most one inversion and $d_l + 1$ multiplications in F. Since $d_l \le m$, the cost of the final step of the algorithm is bounded by one inversion and m multiplications in F.

Adding the cost of the two parts of the algorithm gives us that the total cost of the Euclidean algorithm is at most $m + 2$ inversions and $2nm + n + 2m + 1$ additions and multiplications in F. Therefore, the Euclidean algorithm for $F[x]$ presented in Algorithm 3.2 costs $O(nm)$ arithmetic operations in F.

3.3 The Extended Euclidean Algorithm

Given a Euclidean domain E and two elements $a, b \in E$, the extended Euclidean algorithm (EEA) not only computes $g = \gcd(a, b)$ but also computes $s, t \in E$ satisfying $g = sa+tb$. We obtain g through the exact same steps as in the original Euclidean Algorithm and compute the new output s and t using the quotients from the division steps. Algorithm 3.3 presents a formal description of the EEA. In Example 3.20, we apply the EEA to the polynomials we saw in Example 3.18 over $E = \mathbb{Z}_{17}[x]$.

Algorithm 3.3 Extended Euclidean Algorithm

Input: $a, b \in E$, where E is a Euclidean domain with valuation v
Output: $g = \gcd(a, b), s, t$ such that $g = sa + tb$

1: $r_0 \longleftarrow a; \quad s_0 \longleftarrow 1; \quad t_0 \longleftarrow 0;$
2: $r_1 \longleftarrow b; \quad s_1 \longleftarrow 0; \quad t_1 \longleftarrow 1;$
3: $i = 1;$
4: **while** $r_i \neq 0$ **do**
5: $\quad q_i \longleftarrow r_{i-1}$ quo r_i
6: $\quad r_{i+1} \longleftarrow r_{i-1} - q_i r_i \qquad$ /* $r_{i+1} = r_{i-1}$ rem r_i */
7: $\quad s_{i+1} \longleftarrow s_{i-1} - q_i s_i$
8: $\quad t_{i+1} \longleftarrow t_{i-1} - q_i t_i$
9: $\quad i \longleftarrow i + 1$
10: **end while**
11: $l \longleftarrow i - 1$
12: $v \longleftarrow \mathrm{u}(r_l)^{-1}$
13: $g \longleftarrow v r_l; \quad s \longleftarrow v s_l; \quad t \longleftarrow v t_l$
14: **return** g, s, t

Example 3.20. Let $E = \mathbb{Z}_{17}$, $a(x) = 8x^3 + 3x^2 - 2x - 3$ and $b(x) = 3x^3 - 6x^2 + 6x - 8$. We start the process by setting

$$r_0 \longleftarrow a; \quad s_0 \longleftarrow 1; \quad t_0 \longleftarrow 0$$
$$r_1 \longleftarrow b; \quad s_1 \longleftarrow 0; \quad t_1 \longleftarrow 1.$$

We can write $r_0 = -3r_1 + (2x^2 - x + 7) \pmod{17}$, so

$$\begin{aligned} q_1 &\longleftarrow -3; \\ r_2 &\longleftarrow 2x^2 - x + 7; \\ s_2 &\longleftarrow 1 - (-3)(0) = 1; \\ t_2 &\longleftarrow 0 - (-3)(1) = 3. \end{aligned}$$

Next, since $r_1 = (-7x + 2)r_2 + (6x - 5) \pmod{17}$,

$$\begin{aligned} q_2 &\longleftarrow -7x + 2; \\ r_3 &\longleftarrow 6x - 5; \\ s_3 &\longleftarrow 7x - 2; \\ t_3 &\longleftarrow 4x - 5. \end{aligned}$$

We have $r_2 = (6x + 2)r_3 + 0 \pmod{17}$. Hence

$$\begin{aligned} q_3 &\longleftarrow 6x + 2; \\ r_4 &\longleftarrow 0; \\ s_4 &\longleftarrow -8x^2 - 2x + 5; \\ t_4 &\longleftarrow -7x^2 + 5x - 4. \end{aligned}$$

At this point, $r_4 = 0$, so the while-loop terminates. Finally, the procedure computes $v = \mathrm{u}(r_3)^{-1} \pmod{17} = 3$ and returns

$$g = v \cdot r_3 = x + 2; \quad s = v \cdot s_3 = 4x - 6; \quad t = v \cdot t_3 = -5x + 2.$$

Indeed, a quick check shows $(4x - 6)a(x) + (-5x + 2)b(x) = x + 2$ in $\mathbb{Z}_{17}[x]$, so we have successfully computed the desired $\gcd(a, b)$ and the linear combination $g = sa + tb$.

A useful result of the algorithm is that any remainder in the sequence of r_i can also be expressed as a linear combination of a and b, as presented in Lemma 3.21 below.

Lemma 3.21. *$r_i = s_i a + t_i b$ for $0 \leq i \leq l + 1$.*

Proof. We proceed by induction on i. Initially, $s_0 = 1, s_1 = 0, t_0 = 0$, and $t_1 = 1$, so the base cases $i = 0$ and $i = 1$ hold true with $r_0 = s_0 a + t_0 b = 1 \cdot a + 0 \cdot b = a$ and

$r_1 = s_1a + t_1b = 0 \cdot a + 1 \cdot b = b$. Suppose $1 \leq i \leq l$ and our claim holds true up to i. Then the next iteration of the while-loop defines $q_i = r_{i-1}$ quo r_i, $r_{i+1} = r_{i-1} - q_i r_i$, $s_{i+1} = s_{i-1} - q_i s_i$, and $t_{i+1} = t_{i-1} - q_i t_i$.

Consider $r_{i+1} = r_{i-1} - q_i r_i$. Since $r_{i-1} = s_{i-1}a + t_{i-1}b$ and $r_i = s_i a + t_i b$ by the inductive hypothesis, we have

$$\begin{aligned} r_{i+1} &= r_{i-1} - q_i r_i \\ &= (s_{i-1}a + t_{i-1}b) - q_i(s_i a + t_i b) \\ &= (s_{i-1} - q_i s_i)a + (t_{i-1} - q_i t_i)b \\ &= s_{i+1}a + t_{i+1}b. \end{aligned}$$

Thus we have shown that $r_i = s_i a + t_i b$ for $0 \leq i \leq l + 1$, as claimed. □

Lemma 3.21 proves that the values g, s, and t the algorithm returns at the end correctly satisfy the linear combination $g = sa + tb$: for $i = l$, we have $r_l = s_l a + t_l b$. Then

$$\begin{aligned} g &= \mathrm{n}(r_l) \\ &= r_l / \mathrm{u}(r_l) \\ &= (s_l a + t_l b)/\mathrm{u}(r_l) \\ &= (s_l/\mathrm{u}(r_l))a + (t_l/\mathrm{u}(r_l))b \\ &= sa + tb. \end{aligned}$$

When discussing the result of the EEA, it is convenient to use the following matrix notation.

Definition 3.22. Given the result of the EEA, let

(i) $P_i = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix}$ for $1 \leq i \leq l$, and

(ii) $A_i = P_i P_{i-1} \cdots P_1$, for $1 \leq i \leq l$, with $A_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ for convenience.

For $1 \leq i \leq l$,

$$P_i \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i-1} - q_i r_i \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}.$$

It follows that

$$P_i \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = P_i P_{i-1} \begin{pmatrix} r_{i-2} \\ r_{i-1} \end{pmatrix} = \cdots = P_i P_{i-1} \cdots P_1 \begin{pmatrix} r_0 \\ r_1 \end{pmatrix},$$

and hence

$$P_i P_{i-1} \cdots P_1 \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = A_i \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}.$$

Similarly,

$$A_i \begin{pmatrix} s_0 \\ s_1 \end{pmatrix} = \begin{pmatrix} s_i \\ s_{i+1} \end{pmatrix} \text{ and } A_i \begin{pmatrix} t_0 \\ t_1 \end{pmatrix} = \begin{pmatrix} t_i \\ t_{i+1} \end{pmatrix},$$

from which we can determine the entries of A_i using $s_0 = 1, s_1 = 0, t_0 = 0$, and $t_1 = 1$. That is, $A_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}$ for $0 \leq i \leq l$.

Now, note that $A_l = P_l P_{l-1} \cdots P_1$ can be computed if the quotients q_i in the EA are known. Also,

$$A_l \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r_l \\ 0 \end{pmatrix},$$

where $\mathrm{n}(r_l) = \gcd(a, b)$. This is one of the ingredients for the FEEA.

In the version of the EEA presented in Algorithm 3.3, the remainders r_i are not necessarily unit normal. Consider the case $E = \mathbb{Q}[x]$. The computations following Algorithm 3.3 may produce remainders that have rational coefficients with large numerators and denominators even for relatively small input size. On the other hand, adjusting r_i to be monic helps to keep the coefficients much smaller. Hence, it is beneficial to use a modified version of the extended Euclidean algorithm presented in Algorithm 3.4, wherein ρ_i are used to store the unit parts of the remainders of the division and r_i themselves are unit normal. We will refer to this version of the EEA as the *monic EEA*. The sequences s_i and t_i are appropriately adjusted as well so that Lemma 3.21 still holds true in this new version. The new version of the algorithm also returns l, the number of divisions the extended Euclidean algorithm performs in order to compute the GCD. The fast version of the Euclidean algorithm we introduce in Section 3.4 will use the notations from this version of the EEA.

We define the matrix notations Q_i and B_i for the monic EEA analogously to P_i and A_i introduced in Definition 3.22.

Definition 3.23. Let ρ_i, r_i, and q_i be the result of Algorithm 3.4. Let

Algorithm 3.4 Monic Extended Euclidean Algorithm

Input: $a, b \in E$, where E is a Euclidean domain with valuation v
Output: l, r_i, s_i, t_i for $0 \le i \le l+1$ and q_i for $0 \le i \le l$, where r_i are unit normal

1: $\rho_0 \longleftarrow \mathrm{u}(a)$; $\quad r_0 \longleftarrow \mathrm{n}(a)$; $\quad s_0 \longleftarrow 1$; $\quad t_0 \longleftarrow 0$;
2: $\rho_1 \longleftarrow \mathrm{u}(b)$; $\quad r_1 \longleftarrow \mathrm{n}(b)$; $\quad s_1 \longleftarrow 0$; $\quad t_1 \longleftarrow 1$;
3: $i = 1$;
4: **while** $r_i \neq 0$ **do**
5: $\quad q_i \longleftarrow r_{i-1}$ quo r_i
6: $\quad r_{i+1} \longleftarrow r_{i-1} - q_i r_i$; $\quad s_{i+1} \longleftarrow s_{i-1} - q_i s_i$; $\quad t_{i+1} \longleftarrow t_{i-1} - q_i t_i$
7: $\quad \rho_{i+1} \longleftarrow \mathrm{u}(r_{i+1})$
8: $\quad r_{i+1} \longleftarrow r_{i+1}\rho_{i+1}^{-1}$; $\quad s_{i+1} \longleftarrow s_{i+1}\rho_{i+1}^{-1}$; $\quad t_{i+1} \longleftarrow t_{i+1}\rho_{i+1}^{-1}$
9: $\quad i \longleftarrow i + 1$
10: **end while**
11: $l \longleftarrow i - 1$
12: **return** l, r_i, s_i, t_i for $0 \le i \le l+1$ and q_i for $0 \le i \le l$

(i) $Q_i = \begin{pmatrix} 0 & 1 \\ \rho_{i+1}^{-1} & -q_i\rho_{i+1}^{-1} \end{pmatrix}$ for $1 \le i \le l$, and

(ii) $B_i = Q_i \cdots Q_1$ for $1 \le i \le l$, with $B_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

As with P_i and A_i, we have

$$\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ \rho_{i+1}^{-1} & -q_i\rho_{i+1}^{-1} \end{pmatrix} \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = Q_i \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = Q_i \cdots Q_1 \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \tag{3.3.1}$$

for $1 \le i \le l$. It follows that

$$\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = B_i \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \text{ for } 0 \le i \le l. \tag{3.3.2}$$

As well, $B_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}$ for $0 \le i \le l$, where s_i and t_i come from Algorithm 3.4 with monic r_i, not Algorithm 3.3.

3.3.1 Complexity

We now discuss the complexity of the traditional extended Euclidean algorithm as presented in Algorithm 3.3 for $E = F[x]$. (The scalar multiplications of r_{i+1}, s_{i+1}, and t_{i+1} by $\rho_{i+1}^{-1} \in F$ in Step 8 of Algorithm 3.4 do not increase the cost asymptotically, so the two versions of

the EEA share the same overall complexity.) We will again work with $E = F[x]$. Since the new algorithm is exactly the same as the Euclidean algorithm presented in Algorithm 3.2 except the two line augmentation in the while-loop for s_i and t_i, we need only to find the additional cost of computing the two new sequences for the total cost of the algorithm.

Suppose $f, g \in F[x]$ with $n = \deg f \geq \deg g = m$. To determine the cost of computing s_i and t_i for $2 \leq i \leq l+1$, we need the degrees of the polynomials. First, we need the following lemma.

Lemma 3.24. *Let $d_i = \deg r_i$ for $0 \leq i \leq l$. Then $\deg q_i = d_{i-1} - d_i$ for $1 \leq i \leq l$,*

$$\deg s_i = \sum_{2 \leq j < i} \deg q_j = d_1 - d_{i-1} \text{ for } 2 \leq i \leq l+1, \tag{3.3.3}$$

and

$$\deg t_i = \sum_{1 \leq j < i} \deg q_j = d_0 - d_{i-1} \text{ for } 1 \leq i \leq l+1. \tag{3.3.4}$$

Proof. We will show the proof for (3.3.3) for s_i here. First, we show by induction

$$\deg s_{i-1} < \deg s_i \text{ for } 2 \leq i \leq l+1. \tag{3.3.5}$$

Initially $s_0 = 1, s_1 = 0$, and $s_2 = 1 - q_1 \cdot 0 = 1$, so $-\infty = \deg s_1 < \deg s_2 = 0$ and the base case $i = 2$ is true. Assume the claim has been proven for $2 \leq j \leq i$. At this point, $r_i \neq 0$ and $\deg r_{i-1} = d_{i-1} > d_i = \deg r_i$, so $\deg q_i = \deg r_{i-1} - \deg r_i = d_{i-1} - d_i > 0$. By the inductive hypothesis,

$$\deg s_{i-1} < \deg s_i < \deg s_i + \deg q_i = \deg(s_i q_i).$$

Since $\deg q_i > 0$, it follows that

$$\deg s_i < \deg(q_i s_i) = \deg(s_{i-1} - q_i s_i) = \deg s_{i+1}.$$

Next, we prove (3.3.3) also by induction on i. Since $\deg s_2$ is the trivial sum 0, the base case is true. Suppose then the claim holds true for $2 \leq j \leq i$. By the inductive hypothesis, we have $\deg s_i = \sum_{2 \leq j < i} \deg q_j$, which implies

$$\deg s_{i+1} = \deg(s_i q_i) = \deg s_i + \deg q_i = \sum_{2 \leq j < i} \deg q_j + \deg q_i = \sum_{2 \leq j < i+1} \deg q_j.$$

Since $\deg q_j = d_{j-1} - d_j$, we have

$$\sum_{2 \leq j < i} \deg q_j = (d_1 - d_2) + (d_2 - d_3) + \cdots + (d_{i-2} - d_{i-1}) = d_1 - d_{i-1}.$$

Thus we have shown the equality in (3.3.3) is true. The equality in (3.3.4) for t_i can be proven in the same way. □

We determine the cost of computing t_i for $2 \leq i \leq l+1$ first. To compute $t_{i+1} = t_{i-1} - q_i t_i$, at most $(\deg q_i + 1)(\deg t_i + 1)$ field multiplications are required for the product $q_i t_i$, and subtracting this product from t_{i-1} requires at most $\deg t_{i+1} + 1$ additional field operations. Using (3.3.4), we see the total number of field operations for t_{i+1} for $2 \leq i \leq l$ is

$$\begin{aligned}
&\sum_{2\leq i\leq l} \Big((\deg q_i + 1)(\deg t_i + 1) + (\deg t_{i+1} + 1)\Big) \\
&= \sum_{2\leq i\leq l} (2 \deg q_i \deg t_i + \deg q_i + \deg t_i + \deg t_{i+1} + 2) \\
&= \sum_{2\leq i\leq l} \Big(2(d_{i-1} - d_i)(d_0 - d_{i-1}) + 2(d_0 - d_i + 1)\Big).
\end{aligned}$$

For $i = 1$, computing t_2 requires $n - m + 1$ operations in F. Also, computing t at the end of the algorithm requires $\deg t_l = d_0 - d_{l-1}$.

In the normal case, $l = m + 1$ and $d_i = m - i + 1$, so the total number of field operations in F required to compute the sequence t_i for $2 \leq i \leq l + 1$ and the output t is

$$\begin{aligned}
&(n - m + 1) + \big(n - (m - (m + 1 - 1)) + 1\big) \\
&\qquad + \sum_{2\leq i\leq m+1} \Big(2(n - (m - i + 2)) + 2(n - (m - i + 1) + 1)\Big) \\
&= 2n + m + 2 + 4 \sum_{2\leq i\leq m+1} (n - m + i - 1) \\
&= 2n + m + 2 + 4m(n + m) + 2(m^2 + m) \in O(nm).
\end{aligned}$$

A similar argument shows that computing s_i for $2 \leq i \leq l+1$ and s requires at most $n + 2 + 2(m^2 + m) \in O(m^2)$ arithmetic operations in F. Then computing s_i, t_i, s, and t requires at most $O(nm + m^2) = O(nm)$ arithmetic operations in F. Since the runtime complexity of EA is also $O(nm)$, we see that the runtime complexity of the EEA is $O(nm + nm) = O(nm)$.

3.4 The Fast Extended Euclidean Algorithm

The Fast Extended Euclidean algorithm (FEEA) is a divide-and-conquer algorithm that computes the GCD of two integers or two polynomials over a field. Whereas the Euclidean algorithm sequentially performs a series of polynomial divisions in order to compute the

GCD, the FEEA speeds up this process by bisecting the workload into two recursive processes and using the fact that the leading term of the quotient of the polynomial division is determined solely by the leading terms of the dividend and the divisor. Throughout this section, any reference to the EEA will refer to the monic version in Algorithm 3.4.

Definition 3.25. Let $f = f_n x^n + f_{n-1}x^{n-1} + \cdots + f_0 \in F[x]$ be a polynomial whose leading coefficient f_n is nonzero and $k \in \mathbb{Z}$. Then the *truncated* polynomial is defined as

$$f \upharpoonright k = f \text{ quo } x^{n-k} = f_n x^k + f_{n-1}x^{k-1} + \cdots + f_{n-k},$$

where $f_i = 0$ for $i < 0$. If $k \geq 0$, $f \upharpoonright k$ is a polynomial of degree k whose coefficients are the $k+1$ highest coefficients of f, and if $k < 0$, $f \upharpoonright k = 0$.

Example 3.26. Let $f(x) = 3x^4 + 5x^3 + 7x^2 + 2x + 11$. Then $f \upharpoonright 2 = 3x^2 + 5x + 7$.

Definition 3.27. Let $f, g, f^*, g^* \in F[x]\backslash\{0\}$, $\deg f \geq \deg g$ and $\deg f^* \geq \deg g^*$, and $k \in \mathbb{Z}$. Then we say (f, g) *and* (f^*, g^*) *coincide up to* k if

$$\begin{aligned} f \upharpoonright k &= f^* \upharpoonright k, \text{ and} \\ g \upharpoonright (k - (\deg f - \deg g)) &= g^* \upharpoonright (k - (\deg f^* - \deg g^*)). \end{aligned}$$

It can be shown that this defines an equivalence relation on $F[x] \times F[x]$. Moreover, if (f, g) and (f^*, g^*) coincide up to k and $k \geq \deg f - \deg g$, then $\deg f - \deg g = \deg f^* - \deg g^*$.

Lemma 3.28 ([11], Lemma 11.1)**.** *Let $k \in \mathbb{Z}$ and $f, g, f^*, g^* \in F[x]\backslash\{0\}$. Suppose (f, g) and (f^*, g^*) coincide up to $2k$ and $k \geq \deg f - \deg g \geq 0$. Let $q, r, q^*, r^* \in F[x]$ be the quotients and remainders in the divisions so that $\deg r < \deg g$, $\deg r^* < \deg g^*$, and*

$$\begin{aligned} f &= qg + r \\ f^* &= q^* g^* + r^*. \end{aligned}$$

Then $q = q^*$, and if $r \neq 0$ then either (g, r) and (g^*, r^*) coincide up to $2(k - \deg q)$ or $k - \deg q < \deg g - \deg r$.

Proof. First of all, observe that if $f \upharpoonright 2k = f^* \upharpoonright 2k$, then $x^i f \upharpoonright 2k = x^j f^* \upharpoonright 2k$ for any $i, j \in \mathbb{N}$. Hence, we may safely assume that $\deg f = \deg f^* \geq 2k$ as well as $\deg g = \deg g^*$ by multiplying the pairs (f, g) and (f^*, g^*) by appropriate powers if necessary. Now, we have that $\deg f = \deg f^*$ and $f \upharpoonright 2k = f^* \upharpoonright 2k$, so at least the $2k + 1$ highest terms of

f and f^* are exactly the same. Then $\deg(f - f^*) < \deg f - 2k$. Moreover, we are given $k \geq \deg f - \deg g$, so $\deg f \leq \deg g + k$. It follows that

$$\deg(f - f^*) < \deg f - 2k \leq \deg g - k. \tag{3.4.1}$$

Similarly, from the assumptions that (f, g) and (f^*, g^*) coincide up to $2k$ and $\deg g = \deg g^*$, we have

$$\begin{aligned} \deg(g - g^*) &< \deg g - (2k - (\deg f - \deg g)) = \deg f - 2k \\ &\leq \deg g - k \leq \deg g - \deg q, \end{aligned} \tag{3.4.2}$$

where the last inequality comes from the fact $\deg g = \deg f - \deg g \leq k$. Consider also

$$\deg(r - r^*) \leq \max\{\deg r, \deg r^*\} < \deg g,$$

and note $\deg(f - f^*)$, $\deg(g - g^*)$, and $\deg(r - r^*)$ are all less than $\deg g$. Then, from

$$\begin{aligned} f - f^* &= (qg + r) - (q^*g^* + r^*) - qg^* + qg^* \\ &= q(g - g^*) - (q - q^*)g^* + (r - r^*), \end{aligned} \tag{3.4.3}$$

we get $\deg((q - q^*)g^*) < \deg g = \deg g^*$. It follows that $q - q^* = 0$, or $q = q^*$.

Now, assume $r \neq 0$ and $k - \deg q \geq \deg g - \deg r$. We need to show (g, r) and (g^*, r^*) coincide up to $2(k - \deg q)$, i.e.,

$$g \upharpoonright (2(k - \deg q)) = g^* \upharpoonright (2(k - \deg q))$$
$$r \upharpoonright (2(k - \deg q) - (\deg g - \deg r)) = r^* \upharpoonright (2(k - \deg q) - (\deg g^* - \deg r^*))$$

The first condition is true from the initial assumption (f, g) and (f^*, g^*) coincide up to $2k$. For the second condition, note we have $r - r^* = (f - f^*) - q(g - g^*)$ from (3.4.3) and hence

$$\deg(r - r^*) \leq \max\{\deg(f - f^*), \deg q + \deg(g - g^*)\}.$$

We have $\deg q = \deg f - \deg g$ and know from (3.4.1) and (3.4.2) that $\deg(f - f^*)$ and $\deg(g - g^*)$ are both less than $\deg f - 2k$. Thus we have

$$\deg(r - r^*) < \deg q + \deg f - 2k. \tag{3.4.4}$$

Since we can write $\deg f = \deg q + \deg g$, we find

$$\begin{aligned} \deg(r - r^*) &< \deg q + \deg f - 2k = \deg g - 2(k - \deg q) \\ &= \deg r - (2(k - \deg q) - (\deg g - \deg r)). \end{aligned}$$

So, our claim $r \restriction (2(k - \deg q) - (\deg g - \deg r)) = r^* \restriction (2(k - \deg q) - (\deg g^* - \deg r^*))$ is true if $\deg r = \deg r^*$. From the assumption $k - \deg q \geq \deg g - \deg r$, we have

$$\deg r \geq \deg q + \deg g - k \geq \deg q + \deg f - 2k,$$

which, combined with (3.4.4), shows $\deg r = \deg r^*$. □

Example 3.29. Let

$$f = x^8 + 11x^7 + 3x^5 + 6x + 4, \quad g = x^7 + 6x^5 + 2x^4 + 5x^2 + 1,$$
$$f^* = x^8 + 11x^7 + 3x^5 + 13x^2 + 5x, \quad g^* = x^7 + 6x^5 + 2x^4 + 3x^2 + 7x + 8$$

be polynomials over $\mathbb{Z}_{17}$. We have

$$f \restriction 4 = f^* \restriction 4 = x^4 + 11x^3 + 3x, \quad g \restriction 3 = g^* \restriction 3 = x^3 + 6x + 2,$$

so if we let $k = 2$, then (f, g) and (f^*, g^*) coincide up to $2k = 4$.
Let q, r, q^*, and r^* be the quotients and remainders as in Lemma 3.28. Then

$$q = x + 11, \quad r = 11x^6 + 3x^5 + 12x^4 + 12x^3 + 13x^2 + 5x + 10,$$
$$q^* = x + 11, \quad r^* = 11x^6 + 3x^5 + 12x^4 + 14x^3 + 7x^2 + 5x + 14.$$

We see that $q = q^*$ and $r \restriction 1 = r^* \restriction 1$. Then since $g \restriction 2 = g^* \restriction 2$, (g, r) and (g^*, r^*) coincide up to $2 = 2(k - \deg q)$.

Lemma 3.28 shows us that the quotients in polynomial divisions solely depend on the coefficients of the higher degree terms of the polynomials. That is, if f and f^* share a sufficient number of coefficients for the top terms and g and g^* do as well, the quotients of the divisions $f \div g$ and $f^* \div g^*$ are equal. Moreover, the top terms of the respective remainders will have the same coefficients too, so it is possible the quotients for the divisions $g \div r$ and $g^* \div r^*$ are the same as well.

Recall that $B_l = Q_l \dots Q_1$, where $Q_i = \begin{pmatrix} 0 & 1 \\ \rho_{i+1}^{-1} & -q_i \rho_{i+1}^{-1} \end{pmatrix}$ for $1 \leq i \leq l$. The key idea of the fast GCD algorithm is motivated by this observation: We know that given r_0 and r_1, we can compute $r_l = \gcd(r_0, r_1)$ by computing $\begin{pmatrix} r_l \\ r_{l+1} \end{pmatrix} = B_l \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$. Using Lemma 3.28, we can select some smaller polynomials r_0^* and r_1^* that coincide with r_0 and r_1 to compute the quotients q_i and the leading units ρ_i required to determine B_l. To use this strategy, we must identify exactly how many common quotients we can expect running the EA on the pairs r_0, r_1 and r_0^*, r_1^* to yield.

Example 3.30. Consider the monic polynomials $f, g, f^*, g^* \in \mathbb{Z}_{17}[x]$, where

$$f = x^8 + 6x^6 + 3x^4 + 6x + 6, \quad g = x^7 + x^6 - 5x^5 - 8x^4 - x^3 + 6x^2,$$
$$f^* = x^7 + 6x^5 + 3x^3 + 6, \quad g^* = x^6 + x^5 - 5x^4 - 8x^3 - x^2 + 6x - 2.$$

Let $k = 3$. Then (f, g) and (f^*, g^*) coincide up to $2k = 6$. Applying the EEA on the two polynomial pairs gives us the following result. The first three sets of the quotients and the leading coefficient of the remainders are identical.

i	q_i	q_i^*	ρ_i	ρ_i^*
1	$x-1$	$x-1$	1	1
2	$x+5$	$x+5$	-5	-5
3	$x+7$	$x+7$	4	4
4	$x-3$	x	3	8
5	x^2+5x+7	$x+5$	1	7
6	$x-3$	$x-6$	6	6
7	$x+7$	$x+7$	4	2

Consider the remainder sequences of the Euclidean algorithm applied to two pairs of monic polynomials r_0, r_1 and r_0^*, r_1^* in $F[x]$ with $\deg r_0 > \deg r_1 \geq 0$ and $\deg r_0^* > \deg r_1^* \geq 0$:

$$r_{i-1} = q_i r_i + \rho_{i+1} r_{i+1} \text{ for } 1 \leq i \leq l \text{ with } r_{l+1} = 0$$

and

$$r_{i-1}^* = q_i^* r_i^* + \rho_{i+1}^* r_{i+1}^* \text{ for } 1 \leq i \leq l^* \text{ with } r_{l^*+1} = 0.$$

We define $d_i = \deg r_i$ for $1 \leq i \leq l$, $d_i^* = \deg r_i^*$ for $1 \leq i \leq l^*$, and $d_{l+1} = d_{l^*+1}^* = -\infty$. (By Definition 3.9, we have $\rho_{l+1} = \mathrm{u}(0) = 1$.)

Definition 3.31. For any $k \in \mathbb{Z}_{\geq 0}$, the number $\eta(k) \in \mathbb{N}$ is defined as follows.

$$\eta(k) = \max\{0 \leq j \leq l : \sum_{1 \leq i \leq j} \deg q_i \leq k\}$$

That is, $\eta(k)$ is the number of division steps in the Euclidean algorithm applied to r_0 and r_1 so that the sum of the degrees of the quotients is no more than k.

Recall $\deg q_i = d_{i-1} - d_i$. Then $\sum_{1 \leq i \leq j} \deg q_i = \sum_{1 \leq i \leq j} (d_{i-1} - d_i) = d_0 - d_j$. Thus $\eta(k)$ is uniquely determined by

$$d_0 - d_{\eta(k)} = \sum_{1 \leq i \leq d_{\eta(k)}} \deg q_i \leq k < \sum_{1 \leq i \leq d_{\eta(k)+1}} \deg q_i = d_0 - d_{\eta(k)+1}, \tag{3.4.5}$$

where the second inequality only holds if $\eta(k) < l$. We define $\eta^*(k)$ analogously for r_0^* and r_1^*. The notion $\eta(k)$ plays a key role in the following lemma that describes how two pairs of polynomials that coincide in their higher terms yield the same quotients in the course of the Euclidean algorithm. That is, if the two pairs coincide up to $2k$, then the first $\eta(k)$ quotients coincide.

Lemma 3.32 ([11], Lemma 11.3)**.** *Let* $r_0, r_1, r_0^*, r_1^* \in F[x]$ *be monic with* $\deg r_0 > \deg r_1 \geq 0$ *and* $\deg r_0^* > \deg r_1^* \geq 0$, $k \in \mathbb{N}$, $h = \eta(k)$, *and* $h^* = \eta^*(k)$. *If* (r_0, r_1) *and* (r_0^*, r_1^*) *coincide up to* $2k$, *then*

(i) $\eta(k) = \eta^*(k)$,

(ii) $q_i = q_i^*$ *for* $1 \leq i \leq h$,

(iii) $\rho_{i+1} = \rho_{i+1}^*$ *for* $1 \leq i \leq h-1$.

Remark 3.33. This lemma is incorrectly presented in [11]. The original statement claims $\rho_{h+1} = \rho_{h+1}^*$, but in fact this is not necessarily true. In Example 3.30, $h = h^* = 3$, but $\rho_4 = 3 \neq 8 = \rho_4^*$.

Proof. We will first prove by induction on j the following claim holds for $0 \leq j \leq h-1$:

$$j \leq h^* - 1,\ q_i = q_i^* \text{ and } \rho_{i+1} = \rho_{i+1}^* \text{ for } 1 \leq i \leq j,$$

$$\text{and } (r_j, r_{j+1}) \text{ and } (r_j^*, r_{j+1}^*) \text{ coincide up to } 2(k - \sum_{1 \leq i \leq j} \deg q_i).$$

For the base case $j = 0$, there is nothing to prove. Now, assume the claim holds true for all cases up to $j-1 < h-1$. Then (r_{j-1}, r_j) and (r_{j-1}^*, r_j^*) coincide up to $2(k - \sum_{1 \leq i \leq j-1} \deg q_i)$ by the inductive hypothesis. We apply Lemma 3.28 to get $q_j = q_j^*$, where $r_{j-1} = q_j r_j + \rho_{j+1} r_{j+1}$ and $r_{j-1}^* = q_j^* r_j^* + \rho_{j+1}^* r_{j+1}^*$. Moreover, $j < h \leq l$, so $r_{j+1} \neq 0$. Then either

$$(r_j, r_{j+1}) \text{ and } (r_j^*, r_{j+1}^*) \text{ coincide up to } 2[(k - \sum_{1 \leq i \leq j-1} \deg q_i) - (\deg q_j)] = 2(k - \sum_{1 \leq i \leq j} \deg q_i)$$

or

$$(k - \sum_{1 \leq i \leq j-1} \deg q_i) - \deg q_j = k - \sum_{1 \leq i \leq j} \deg q_i < d_j - d_{j+1} = \deg q_{j+1},$$

the latter of which is equivalent to $k < \sum_{1 \leq i \leq j+1} \deg q_i$. But $j < h$, and hence

$$\sum_{1 \leq i \leq j+1} \deg q_i \leq \sum_{1 \leq i \leq h} \deg q_i \leq k.$$

So (r_j, r_{j+1}) and (r_j^*, r_{j+1}^*) must coincide up to $2(k - \sum_{1\le i\le j} \deg q_i)$. Then we have

$$r_{j+1} \upharpoonright (2(k - \sum_{1\le i\le j} \deg q_i) - (d_j - d_{j+1})) = r_{j+1}^* \upharpoonright (2(k - \sum_{1\le i\le j} \deg q_i^*) - (d_j^* - d_{j+1}^*)),$$

and since $k - \sum_{1\le i\le j+1} \deg q_i \ge 0$, we have that the two truncated polynomials are nonzero and that r_{j+1} and r_{j+1}^* must have the same leading coefficient. That is, $\rho_{j+1} = \rho_{j+1}^*$. The assertion $j \le h^* - 1$ is true because $\sum_{1\le i\le j} \deg q_i^* = \sum_{1\le i\le j} \deg q_i < \sum_{1\le i\le h} \deg q_i \le k$.

Finally, (r_{h-1}, r_h) and (r_{h-1}^*, r_h^*) coincide up to $2(k - \sum_{1\le i\le h-1} \deg q_i)$, so we can apply Lemma 3.28 again to obtain $q_h = q_h^*$. It follows that $h = \eta(k) = \eta^*(k) = h^*$.

□

The lemma refines the strategy for computing the GCD of two polynomials of very high degrees: given large monic polynomials $r_0, r_1 \in F[x]$ with $\deg r_0 > \deg r_1$, it is sufficient to consider a pair of smaller polynomials (r_0^*, r_1^*) that coincide with (r_0, r_1) up to $2k$ for some $k > \deg r_0 - \deg r_1$. Lemma 3.32 guarantees that the first $\eta(k)$ quotients q_i as well as the first $\eta(k) - 1$ leading coefficients of the remainders ρ_i we compute will be the same for both pairs. After computing the first group of quotients in this way, we can compute the rest of the quotients in the sequence similarly using $r_{\eta(k)}$ and $r_{\eta(k)+1}$.

Adapting this strategy, the fast algorithm makes two recursive calls and a single division to compute the quotients $q_1, \ldots, q_l$ and the leading coefficients $\rho_2, \ldots, \rho_l$ of the remainders $r_2, \ldots, r_l$. As well, rather than returning the sequence of quotients q_i, the algorithm returns the matrix B_l, the product of the corresponding matrices Q_i, i.e., $B_l = Q_l \cdots Q_1$, where $B_l \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} r_l \\ r_{l+1} \end{pmatrix}$. That is, the algorithm makes one recursive call computing $R = Q_{\eta(k)-1} \cdots Q_1$, one division $r_{\eta(k)-1}$ rem $r_{\eta(k)}$ for $Q_{\eta(k)}$, and a second recursive call computing $S = Q_l \cdots Q_{\eta(k)+1}$ and returns $B_l = SQ_{\eta(k)}R$.

The use of $\eta(k)$ addresses the problem of deciding how to divide the division steps into two smaller tasks of roughly the same size. An obvious approach to divide the total number of steps l is to group them into two equal parts so that the algorithm first computes the first $l/2$ of the quotients and then the second half. However, if there are some quotients with much higher degrees than some others, this may not be the optimal choice. Therefore, the right choice is to divide the sequence of quotients into two parts so that the sums of the degrees of the quotients are roughly the same size as the other. Note that in the case where the degree sequence of remainders is normal, the two approaches yield the same result.

The Fast Extended Euclidean Algorithm is presented in Algorithm 3.5. Given the input $r_0, r_1 \in F[x]$, both monic, and $k \in \mathbb{N}$, the algorithm returns two parameters, $h = \eta(k)$ and B_h, where the integer h is the number of division steps performed and $B_h \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} r_h \\ r_{h+1} \end{pmatrix}$.

3.4.1 Proof of Correctness

We will briefly discuss the correctness of the values returned by the algorithm. The overall correctness of the rest of the algorithm follows from applying induction over k and Lemma 3.32, assuming that the returns from the recursive calls are correct.

Initially in Step 1, the algorithm checks if $r_1 = 0$ or $k < d_0 - d_1$, and if true, the correct values $l = 0$ and $B_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ are returned.

In Step 4, the algorithm obtains through a recursive call $j = \eta^*(k_1)+1$ and $R^* = B^*_{j-1} = Q^*_{j-1}Q^*_{j-2}\cdots Q^*_1$ for (r^*_0, r^*_1), which is constructed in Step 3 to coincide with (r_0, r_1) up to $2k_1$. From Lemma 3.32, we have $q_i = q^*_i$ for $1 \le i \le j-1$ and $\rho_i = \rho^*_i$ for $2 \le i \le j-1$. As well, ρ^*_j and ρ_j are not necessarily equal. So we have

$$R^* = Q^*_{j-1}B_{j-2} = \begin{pmatrix} 0 & 1 \\ \rho_j^{*-1} & -q_{j-1}\rho_j^{*-1} \end{pmatrix} \begin{pmatrix} s_{j-2} & t_{j-2} \\ s_{j-1} & t_{j-1} \end{pmatrix} = \begin{pmatrix} s_{j-1} & t_{j-1} \\ (\rho_j/\rho^*_j)s_j & (\rho_j/\rho^*_j)t_j \end{pmatrix}$$

and

$$R^* \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} r_{j-1} \\ (\rho_j/\rho^*_j)r_j \end{pmatrix}.$$

Let $\tilde{r}_j = (\rho_j/\rho^*_j)r_j = \rho_j^{*-1}(\rho_j r_j)$. Then $\tilde{\rho}_j = \mathrm{lc}(\tilde{r}_j) = \rho_j^{*-1}\rho_j$, and we find

$$R = B_{j-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1/\tilde{\rho}_j \end{pmatrix} R^*, \text{ and } r_j = \tilde{r}_j/\tilde{\rho}_j,$$

where r_j is monic.

We have $d_0 - d_{j-1} \le k_1 < d_0 - d_j$ from the inequality in (3.4.5). So, if $k < d_0 - d_j$, then $d_0 - d_{j-1} \le k_1 = \lfloor k/2 \rfloor \le k < d_0 - d_j$, and indeed $\eta(k) = j-1$, which the algorithm returns alongside the matrix R. Therefore the algorithm returns the correct values in Step 6. On the other hand, if $d_0 - d_j \le k$ then $\eta(k) \ge j$; the algorithm continues on to compute q_j and r_{j+1} in Step 7, which gives the matrix Q_j, defined in Step 8. Note that Step 7 is where the algorithm performs the divisions to determine the quotients q_i and units ρ_i required to compute the return parameter $B_h = Q_h \cdots Q_1$.

Algorithm 3.5 Fast Extended Euclidean Algorithm

Input: monic polynomials $r_0, r_1 \in F[x], n = d_0 = \deg r_0 > d_1 = \deg r_1$, and $k \in \mathbb{N}$ with $d_0/2 \le k \le d_0$

Output: $h = \eta(k) \in \mathbb{Z}_{\ge 0}$ as in Definition 3.31 and $M = B_h = Q_h Q_{h-1} \cdots Q_1 \in F[x]^{2\times 2}$

1: **if** $r_1 = 0$ or $k < d_0 - d_1$ **then return** 0 and $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

/* **Compute** $j = \eta(k/2)$ **and** $R = Q_{j-1} \cdots Q_1$ */

2: $k_1 \longleftarrow \lfloor k/2 \rfloor$

3: $r_0^* \longleftarrow r_0 \upharpoonright 2k_1$; $\quad r_1^* \longleftarrow r_1 \upharpoonright (2k_1 - (d_0 - d_1))$

4: call the algorithm recursively with r_0^*, r_1^* and k_1 to obtain $j - 1 = \eta^*(k_1)$ and $R^* = Q_{j-1}^* Q_{j-2} \cdots Q_1$, where $Q_{j-1}^* = \begin{pmatrix} 0 & 1 \\ \rho_j^{*-1} & -q_{j-1}\rho_j^{*-1} \end{pmatrix}$ and $\begin{pmatrix} r_{j-1}^* \\ r_j^* \end{pmatrix} = R^* \begin{pmatrix} r_0^* \\ r_1^* \end{pmatrix}$

5: $\begin{pmatrix} r_{j-1} \\ \tilde{r}_j \end{pmatrix} \longleftarrow R^* \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$; $\quad \tilde{\rho}_j \longleftarrow \operatorname{lc}(\tilde{r}_j)$;
$R \longleftarrow \begin{pmatrix} 1 & 0 \\ 0 & \tilde{\rho}_j^{-1} \end{pmatrix} R^*$;
$r_j \longleftarrow \tilde{\rho}_j^{-1} \tilde{r}_j$; $\quad d_j \longleftarrow \deg r_j$

6: **if** $r_j = 0$ or $k < d_0 - d_j$ **then return** $j - 1$ and R

/* **Compute** q_j **and** Q_j */

7: $q_j \longleftarrow r_{j-1}$ quo r_j; $\quad \rho_{j+1} \longleftarrow \operatorname{lc}(r_{j-1}$ rem $r_j)$;
$r_{j+1} \longleftarrow (r_{j-1}$ rem $r_j)\rho_{j+1}^{-1}$; $\quad d_{j+1} \longleftarrow \deg r_{j+1}$

8: $Q_j \longleftarrow \begin{pmatrix} 0 & 1 \\ \rho_{j+1}^{-1} & -q_j\rho_{j+1}^{-1} \end{pmatrix}$

/* **Compute** $h = \eta(k)$ **and** $S = Q_h \cdots Q_{j+1}$ */

9: $k_2 \longleftarrow k - (d_0 - d_j)$

10: $r_j^* \longleftarrow r_j \upharpoonright 2k_2$; $\quad r_{j+1}^* \longleftarrow r_{j+1} \upharpoonright (2k_2 - (d_j - d_{j+1}))$

11: call the algorithm recursively with r_j^*, r_{j+1}^* and k_2 to obtain $h - j = \eta^*(k_2)$ and $S^* = Q_h^* Q_{h-1} \cdots Q_{j+1}$, where $Q_h^* = \begin{pmatrix} 0 & 1 \\ \rho_{h+1}^{*-1} & -q_h\rho_{h+1}^{*-1} \end{pmatrix}$ and $\begin{pmatrix} r_h^* \\ r_{h+1}^* \end{pmatrix} = S^* \begin{pmatrix} r_j^* \\ r_{j+1}^* \end{pmatrix}$

12: $\begin{pmatrix} r_h \\ \tilde{r}_{h+1} \end{pmatrix} \longleftarrow S^* \begin{pmatrix} r_j \\ r_{j+1} \end{pmatrix}$; $\quad \tilde{\rho}_{h+1} \longleftarrow \operatorname{lc}(\tilde{r}_{h+1})$;
$S \longleftarrow \begin{pmatrix} 1 & 0 \\ 0 & \tilde{\rho}_{h+1}^{-1} \end{pmatrix} S^*$

13: $M \longleftarrow SQ_jR$

14: **return** h and M

From the second recursive call in Step 11, we obtain $S^* = Q_h^* Q_{h-1} \cdots Q_{j+1}$. We compute $S = Q_h \cdots Q_{j+1}$ in the same way we found R using R^*:
We have

$$S^* \begin{pmatrix} r_j \\ r_{j+1} \end{pmatrix} = S^* B_j \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = Q_h^* Q_{h-1} \cdots Q_1 \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$$
$$= Q_h^* B_{h-1} \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = Q_h^* \begin{pmatrix} r_{h-1} \\ r_h \end{pmatrix} = \begin{pmatrix} r_h \\ (\rho_{h+1}/\rho_{h+1}^*) r_{h+1} \end{pmatrix}.$$

We define $\tilde{r}_j = \rho_{h+1}^{*-1}(\rho_{h+1} r_{h+1})$. Then $\tilde{\rho}_{h+1} = \mathrm{lc}(\tilde{r}_{h+1}) = \rho_{h+1}^{*-1}\rho_{h+1}$, and we can now compute

$$S = \begin{pmatrix} 1 & 0 \\ 0 & 1/\tilde{\rho}_{h+1} \end{pmatrix} S^* = Q_h \cdots Q_{j+1} \text{ and } r_{h+1} = \tilde{r}_{h+1}/\tilde{\rho}_{h+1}.$$

Now, the value $h - j$ also obtained in Step 11 implies

$$d_j - d_h = (d_0 - \sum_{1 \le i \le j} m_i) - (d_0 - \sum_{1 \le i \le h} m_i) = \sum_{j+1 \le i \le h} m_i \le k_2 < d_j - d_{h+1},$$

or $h = l$. But then adding $(d_0 - d_j)$ to each part of the inequality gives

$$d_0 - d_h = (d_0 - d_j) + (d_j - d_h) \le (d_0 - d_j) + k_2 < (d_0 - d_j) + (d_j - d_{h+1}) = d_0 - d_{h+1}$$

or $h = l$. Then from the equality $(d_0 - d_j) + k_2 = k$ and (3.4.5), we have
$\eta((d_0 - d_j) + k_2) = \eta(k) = h$.

Finally, consider the values h and SQ_jR that are returned in Step 14. Since $S = Q_h \cdots Q_{j+1}$ and $R = Q_{j-1} \cdots Q_1$, $M = SQ_jR = Q_h \cdots Q_1 = B_h$. Therefore the algorithm correctly returns $h = \eta(k)$ and B_h for the input r_0, r_1 and k.

Remark 3.34. The FEEA requires the input polynomials f and g to be monic with $\deg f > \deg g \ge 0$. Given f and g that do not satisfy these conditions, we modify the input as follows.

Step 1a. if $\deg f = \deg g$ and $f/\mathrm{lc}(f) = g/\mathrm{lc}(g)$, return $g/\mathrm{lc}(g)$.

Step 1b. if f and g are monic with $\deg f = \deg g$:
Let $\rho_2 = \mathrm{lc}(f - g), r_0 = g$ and $r_1 = (f - g)/\rho_2$ and call the FEEA with these

three parameters, which returns h and $B_h = \begin{pmatrix} s_h & t_h \\ s_{h+1} & t_{h+1} \end{pmatrix}$ as results. Then $r_h = s_h g + t_h(f-g)/\rho_2 = (t_h/\rho_2)f + (s_h - t_h/\rho_2)g$, so we compute the matrix

$$R = B_h \begin{pmatrix} 0 & 1 \\ \rho_2^{-1} & -\rho_2^{-1} \end{pmatrix} = \begin{pmatrix} t_h/\rho_2 & s_h - t_h/\rho_2 \\ t_{h+1}/\rho_2 & s_{h+1} - t_{h+1}/\rho_2 \end{pmatrix}.$$

Then the top entry of the vector $R \begin{pmatrix} f \\ g \end{pmatrix}$ is r_h.

Step 1c. if $\deg f > \deg g$ but f and g are not monic:
Let $r_0 = f/\operatorname{lc}(f)$ and $r_1 = g/\operatorname{lc}(g)$ and call the FEEA to obtain h and B_h. Divide the first and second rows of B_h by $\operatorname{lc}(f)$ and $\operatorname{lc}(g)$ respectively and denote the resulting matrix R. Then the top entry of the vector $R \begin{pmatrix} f \\ g \end{pmatrix}$ is r_h.

Remark 3.35. The FEEA requires $d_0/2 \leq k \leq d_0$. If given $0 < k < d_0/2$, it is sufficient to call the FEEA with $r_0 \upharpoonright 2k, r_1 \upharpoonright (2k - (\deg r_0 - \deg r_1))$ and k. Apply the same corrections as in Step 4 of the algorithm to the output.

It is possible to use Algorithm 3.5 to compute any single row r_h, s_h, t_h of the EEA for $1 \leq h \leq l$, keeping in mind the adjustment for $0 < h < d_0/2$ described in Remark 3.35. We can specify h by selecting a $k \in \mathbb{N}$ so that $\deg r_0 - k$ is the lower bound on $\deg r_h$ or, equivalently, k an upper bound on $\sum_{1 \leq i \leq h} \deg q_i$ so that $h = \eta(k)$. In particular, if we use $k = d_0$, then the return values will be $\eta(d_0) = l$ and $B_l = Q_l \cdots Q_1 = \begin{pmatrix} s_l & t_l \\ s_{l+1} & t_{l+1} \end{pmatrix}$. That is, given $f, g \in F[x]$, we can find the $r_l = \gcd(f, g)$ by running the algorithm with f, g, and $k = \deg f$ to obtain the matrix M and then computing the top row of the matrix-vector product $M \begin{pmatrix} f \\ g \end{pmatrix}$.

3.4.2 Complexity

Let us now consider the cost of running the FEEA. Let $T(k)$ denote the number of arithmetic operations in F that the algorithm uses on input k. Step 4 takes $T(k_1) = T(\lfloor k/2 \rfloor)$ operations, and Step 11 takes $T(k_2)$ operations. Since $k_2 = k - (d_0 - d_j) < k - k_1 = \lceil k/2 \rceil$, $k_2 \leq \lfloor k/2 \rfloor$ and $T(k_2) \leq T(\lfloor k/2 \rfloor)$. Thus the two steps take a total of at most $2T(\lfloor k/2 \rfloor)$ arithmetic operations in F.

Next, we consider the cost of making the corrections in Step 5. The degrees of the entries of the matrix $R^* = \begin{pmatrix} s_{j-1} & t_{j-1} \\ (\rho_j/\rho_j^*)s_j & (\rho_j/\rho_j^*)t_j \end{pmatrix}$ are $d_1 - d_{j-2}, d_0 - d_{j-2}, d_1 - d_{j-1}$, and $d_0 - d_{j-1}$. Since $j = \eta(k_1)$, we have $d_0 - d_{j-1} \leq k_1 = \lfloor k/2 \rfloor$ and thus all entries of R^* have degrees at most $\lfloor k/2 \rfloor$. As well, $d_0/2 \leq k \leq d_0$, so $d_1 < d_0 \leq 2k$. The matrix-vector multiplication $R^* \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$ therefore requires four multiplications of polynomials of degrees at most $\lfloor k/2 \rfloor$ by polynomials of degree at most $2k$. By dividing the larger polynomials into blocks of degrees at most $\lfloor k/2 \rfloor$, these multiplications can be performed in $16M(\lfloor k/2 \rfloor) + O(k) \leq 8M(k) + O(k)$ operations in F, where $M(n)$ denotes the cost of multiplying two polynomials of degrees at most n. (This can be further optimized to be computed in $4M(k) + O(k)$ operations, as outlined in [22]. This approach requires an extra return parameter $\begin{pmatrix} r_{j-1}^* \\ r_j^* \end{pmatrix}$ and uses the equality $R^* \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = R^* \begin{pmatrix} r_0 - r_0^* x^{d_0 - 2k_1} \\ r_1 - r_1^* x^{d_0 - 2k_1} \end{pmatrix} = \begin{pmatrix} r_{j-1}^* r_0^* x^{d_0 - 2k_1} \\ r_j^* r_0^* x^{d_0 - 2k_1} \end{pmatrix}$.)

Computing R from R^* can be achieved in $O(k)$ arithmetic operations by scaling the second row of R^* rather than through a matrix-matrix multiplication, since the top rows of the two matrices are the same and the second rows differ by a factor of $1/\tilde{\rho}_j$. Given the degrees of the elements of R^* discussed earlier, multiplying the two elements of R^* by a scalar costs at most $\lfloor k/2 \rfloor$ for each. Since r_j is of degree $d_j < d_0 \leq 2k$, computing $r_j = \tilde{r}_j/\tilde{\rho}_j$ costs at most k multiplications. Hence the total cost of Step 5 is at most $8M(k) + O(k)$ operations in F.

We can find the cost of Step 12 similarly. The elements of the matrix S^* are of degrees $d_{j+1} - d_{h-1}, d_j - d_{h-1}, d_{j+1} - d_h$, and $d_j - d_h$, which are at most $d_j - d_h \leq k_2 \leq \lfloor k/2 \rfloor$, and $d_{j+1} < d_j < 2k$. Thus the matrix-vector multiplication costs at most $8M(k) + O(k)$ operations in F. Moreover, computing S using S^* costs at most $O(k)$ operations in F. Thus the whole step costs $8M(k) + O(k)$ operations in F.

In Step 7, we divide r_{j-1} by r_j to obtain the quotient q_j and the remainder r_{j+1}. The divisor r_j has degree $d_j < 2k$. The degree of q_j is $d_{j-1} - d_j$. By this step, from the two return criteria, we have $k_1 < d_0 - d_j \leq k$ and

$$0 < d_0 - k \leq d_j < d_{j+1} \leq d_0 \leq 2k.$$

Thus $0 < \deg q_j = d_{j-1} - d_j \leq d_0 - (d_0 - k) = k$, although in practice we often have $\deg q_j = 1$. A quotient of degree at most k can be computed using $4M(k) + O(k)$ operations

in F and the remainder r_{j+1} in $2M(k)+O(k)$ operations in F using fast algorithms such as one presented in [11], Algorithm 9.5. So the division costs at most $6M(k)+O(k)$ operations in F. If the remainder sequence is normal and $\deg q_j = 1$ for all j, then the division costs $O(k)$.

We consider the cost of computing $M = SQ_jR$ in Step 13 in two stages. First, consider the cost of computing $B_j = Q_jR = \begin{pmatrix} 0 & 1 \\ \rho_{j+1}^{-1} & -q_j\rho_{j+1}^{-1} \end{pmatrix} \begin{pmatrix} s_{j-1} & t_{j-1} \\ s_j & t_j \end{pmatrix} = \begin{pmatrix} s_j & t_j \\ s_{j+1} & t_{j+1} \end{pmatrix}$. The first row of B_j is exactly the same as the second row of $R = B_{j-1}$. Therefore, we only need to compute two new elements, $s_{j+1} = (s_{j-1} - q_j s_j)/\rho_{j+1}$ and $t_{j+1} = (t_{j-1} - q_j t_j)/\rho_{j+1}$. Since s_j and t_j are of degrees at most $\lfloor k/2 \rfloor$ and q_j is at most of degree k, computing these elements costs at most $2M(k)+O(k)$ operations in F. Then computing SB_j costs at most $6M(k)+O(k)$ operations in F, since the degrees of the entries of S are at most $\lfloor k/2 \rfloor$ and the degrees of the entries of B_j are at most $\lfloor k/2 \rfloor$ for the top row and k for the second row. Hence the total cost of computing M in Step 13 is at most $8M(k)+O(k)$ operations in F. (We could implement Strassen's algorithm for matrix multiplication, as outlined in Algorithm 12.1 in [11] to reduce the cost to $7M(k)+O(k)$.)

There are at most three inversions required $\rho_j^{*-1}, \tilde{\rho}_{j+1}^{*-1}$, and $\tilde{\rho}_{h+1}^{*-1}$ per recursive layer. These can be computed in the total of $3k$ operations in F for the entire recursive process and are asymptotically insignificant in the overall cost of the algorithm.

Finally, we can add the costs for all parts of the algorithm and compute the total cost of running Algorithm 3.5. We see that T satisfies the recursive inequalities

$$T(0) = 0 \text{ and } T(k) \leq 2T(\lfloor k/2 \rfloor) + 30M(k) + ck \text{ for } k > 0,$$

for some constant $c \in \mathbb{R}$. Hence we conclude that

$$T(k) \leq (30M(k) + O(k)) \log k \in O(M(k) \log k).$$

Now, $n = d_0$ and $d_0/2 \leq k \leq d_0$, so $k \in O(n)$. Thus we have $O(M(k)\log k) \in O(M(n)\log n)$. The adjustments discussed in Remarks 3.34 and 3.35 can each be made in $O(n)$ operations in F and do not affect the algorithm's overall cost of $O(M(n)\log n)$. We used Karatsuba's multiplication algorithm in our implementation of the FEEA, which makes $M(n) = O(n^{\log_2 3})$ and the running cost of FEEA $O(n^{\log_2 3}\log n)$ arithmetic operations in F.

3.5 Timings

Here we compare the performance of the Euclidean Algorithm against the Fast Extended Euclidean Algorithm. Each algorithm was implemented in C and uses a Maple interface. All reported timings are in CPU seconds and were obtained using Maple's `time` routine. All tests were executed using Maple 16 on a 64 bit AMD Opteron 150 CPU 2.4 GHz with 2 GB memory running Linux. We measure the CPU time of each of the test cases.

The following Maple code was used to generate ten sets of two random dense polynomials $f, g \in \mathbb{Z}_p$ for the GCD computations of degree $d = 1000 \cdot 2^i$, $0 \leq i \leq 9$.

```
> f := modp1( Randpoly( d, x ), p );
> g := modp1( Randpoly( d, x ), p );
```

We used a 30-bit prime $p = 1073741789$. Given f and g, the FEEA applied the adjustments discussed in Remark 3.34 so that $\deg r_0 > \deg r_1$ and r_0 and r_1 are monic. We used $k = \deg r_0$. The timings are presented in Table 3.1.

Table 3.1: EA vs. FEEA (Cutoff = 150)

i	d	EA (ms)	FEEA (ms)	EA/FEEA
0	1000	29.1	31.0	0.938
1	2000	86.2	88.2	0.977
2	4000	341.4	254.3	1.341
3	8000	1371.4	743.4	1.845
4	16000	5544.8	2208.2	2.511
5	32000	22417.6	6552.0	3.421
6	64000	90585.2	19529.0	4.638
7	128000	368053.0	59711.0	6.164
8	256000	1495268.0	178750.0	8.365
9	512000	6050936.0	555564.0	10.892

We have implemented FEEA so that for truncated polynomials of degrees less than a preset cutoff level, the computation uses EEA instead. This strategy avoids the heavy overhead cost of FEEA and takes advantage of the efficiency of the straightforward EEA for low degree polynomials. In our testing, we set the cut-off level to be 150.

Initially, the FEEA shows no significant performance improvement over the EA. This is due to the overhead cost for all the extra computation such as s_i and t_i in the base cases as well as the matrix and vector multiplications the FEEA requires. However, the FEEA

quickly proves to be more efficient than the EA. By the time $d = 512000$, the FEEA is almost 11 times faster than the EA.

3.6 Application: Resultant Computation

The resultant of two polynomials is a useful tool in many areas of mathematics such as rational function integration and algebraic geometry, and like the GCD, it can be computed using the polynomial remainder sequence. In this section, we will examine how the FEEA can be modified to efficiently compute the resultant.

Definition 3.36. Let F be a field. Let

$$f = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 \in F[x]$$

with roots $\alpha_1, \ldots, \alpha_n$ and

$$g = b_m x^m + b_{m-1} x^{m-1} + \cdots + b_0 \in F[x]$$

with roots $\beta_1, \ldots, \beta_m$. The *resultant* of f and g is defined to be

$$\operatorname{res}(f, g) = a_n^m b_m^n \prod_{i=1}^{n} \prod_{j=1}^{m} (\alpha_i - \beta_j).$$

It is clear from the definition that if f and g share a common root, then $\operatorname{res}(f, g) = 0$. On the other hand, if $\operatorname{res}(f, g) = 0$, then it must be true that $\prod_{i=1}^{n} \prod_{j=1}^{m} (\alpha_i - \beta_j) = 0$. It follows that $\alpha_i = \beta_j$ for some i and j, i.e., f and g have a common root.

Remark 3.37. The resultant is traditionally defined as the determinant of the *Sylvester matrix* of f and g (See [9]).

Lemma 3.38. *Let $f, g \in F[x]$, F a field, where $f = a_n \prod_{i=1}^{n} (x - \alpha_i)$ and $g = b_m \prod_{i=1}^{m} (x - \beta_i)$. Then we have the following properties of the resultant.*

(i) *if $f = c$ for some $c \in F$, then* $\operatorname{res}(f, g) = c^m$

(ii) $\operatorname{res}(f, g) = (-1)^{nm} \operatorname{res}(g, f)$

(iii) $\operatorname{res}(f, g) = a_n^m \prod_{i=1}^{n} g(\alpha_i) = (-1)^{nm} b_m^n \prod_{j=1}^{m} f(\beta_j)$

(iv) *if $h \in F[x]$,* $\operatorname{res}(f, g \cdot h) = \operatorname{res}(f, g) \cdot \operatorname{res}(f, h)$

(v) if $c \in F$, $\operatorname{res}(f, cg) = c^n \operatorname{res}(f, g)$.

These properties follow directly from the definition. We will show here the proof of (iii).

Proof of Part (iii). We have $g(\alpha_i) = b_m \prod_{j=1}^{m}(\alpha_i - \beta_j)$ for $1 \le i \le n$, so

$$\begin{aligned}\operatorname{res}(f, g) &= a_n^m b_m^n \prod_{i=1}^{n}\prod_{j=1}^{m}(\alpha_i - \beta_j) \\ &= a_n^m \prod_{i=1}^{n}\left(b_m \prod_{j=1}^{m}(\alpha_i - \beta_j)\right) \\ &= a_n^m \prod_{i=1}^{n} g(\alpha_i).\end{aligned}$$

Similarly, we use the fact $f(\beta_j) = a_n \prod_{i=1}^{n}(\beta_j - \alpha_i) = a_n(-1)^n \prod_{i=1}^{n}(\alpha_i - \beta_j)$ and obtain the second equality $\operatorname{res}(f, g) = b_m^n \prod_{j=1}^{m}\left(a_n(-1)^n \prod_{i=1}^{n}(\alpha_i - \beta_j)\right) = (-1)^{nm} b_m^n \prod_{j=1}^{m} f(\beta_j)$. □

Theorem 3.39. *Let* $f, g \in F[x]\backslash\{0\}$, *where* $f = a_n \prod_{i=1}^{n}(x - \alpha_i)$, $g = b_m \prod_{i=1}^{m}(x - \beta_i)$, *and* $m, n > 0$. *Suppose* $r, q \in F[x]$ *and* $r = f \text{ rem } g$ *and* $q = f \text{ quo } g$ *so that* $f = gq + r$. *Then*

$$\operatorname{res}(f, g) = \begin{cases} 0, & \text{if } r = 0 \\ (-1)^{nm} b_m^{n-l} \operatorname{res}(g, r), \text{ where } l = \deg r \ge 0, & \text{if } r \ne 0. \end{cases}$$

Proof. If $r = 0$, $f = qg$ and f and g share a common root and we get $\operatorname{res}(f, g) = 0$. Now, suppose $r \ne 0$. By Lemma 3.38 (iii),

$$\operatorname{res}(f, g) = (-1)^{nm} b_m^n \prod_{j=1}^{m} f(\beta_j).$$

But $f(\beta_j) = g(\beta_j)q(\beta_j) + r(\beta_j)$ and $g(\beta_j) = 0$ for all j, so $f(\beta_j) = r(\beta_j)$. Then

$$\operatorname{res}(f, g) = (-1)^{nm} b_m^n \prod_{j=1}^{m}(g(\beta_j)q(\beta_j) + r(\beta_j)) = (-1)^{nm} b_m^n \prod_{j=1}^{m} r(\beta_j).$$

Now, $\operatorname{res}(r, g) = (-1)^{ml} b_m^l \prod_{j=1}^{m} r(\beta_j)$ again by Lemma 3.38 (iii). Thus we can write

$$\operatorname{res}(f, g) = (-1)^{nm-ml} b_m^{n-l}\left((-1)^{ml} b_m^l \prod_{j=1}^{m} r(\beta_j)\right) = (-1)^{nm-ml} b_m^{n-l} \operatorname{res}(r, g).$$

Finally, $\operatorname{res}(r, g) = (-1)^{ml} \operatorname{res}(g, r)$, so $\operatorname{res}(f, g) = (-1)^{nm} b_m^{n-l} \operatorname{res}(g, r)$. □

Theorem 3.39 tells us that we can use the remainder sequence from the EA to compute the resultant of two polynomials. This observation motivates the recursive algorithm, as presented in Algorithm 3.6, that uses the framework of the classical Euclidean GCD algorithm.

Algorithm 3.6 RES(f,g)

Input: $f, g \in F[x]\backslash\{0\}$ with $n = \deg f$, $m = \deg g$, and $\mathrm{lc}(g) = b$
Output: $\mathrm{res}(f, g)$

1: **if** $n = 0$ **then return** f^m
2: **if** $m = 0$ **then return** g^n
3: **if** $n < m$ **then return** $(-1)^{nm}\mathrm{RES}(g, f)$
4: $r \longleftarrow f \text{ rem } g$; $\quad l \longleftarrow \deg r$
5: **if** $r = 0$ **then return** 0
6: **return** $(-1)^{nm} b^{n-l}\mathrm{RES}(g, r)$

Let us briefly consider the cost of Algorithm 3.6. The total cost of the algorithm is essentially the cost of the classical Euclidean algorithm, $O(nm)$ arithmetic operations in F, plus the cost of computing $(-1)^{nm}b^{n-l}$ and multiplying it with $\mathrm{res}(g, r)$ in each recursive layer (or f^m or g^n in the base case). If $m > n$, Step 3 computes $\mathrm{res}(g, f)$ instead and multiplies it by $(-1)^{nm}$ at the cost of $O(1)$ additional work, so we can assume $n \geq m$.

Computing $(-1)^{nm}b^{n-l}$ for all recursive layers requires at most $O(n)$ arithmetic operations in F in total: Computing b^{n-l} costs $O(\log(n - l))$ arithmetic operations in F. If there are many recursive layers then each $n - l$ is small and $O(\log(n - l))$ close to $O(1)$. The maximum length of remainder sequence is $m + 1$ when the sequence is normal, so the total cost is $O(m) \in O(n)$ arithmetic operations in F. On the other hand, if $n - l$ are large, the exponentiation b^{n-l} costs $O(\log(n-l)) \in O(\log n)$ arithmetic operations in F and we only have a small number of recursive calls. Hence the total cost of computing b^{n-l} is $O(\log n) \in O(n)$ arithmetic operations in F.

Computing and multiplying each $(-1)^{nm}$ costs $O(1)$ arithmetic operations in F and does not contribute significantly to the overall complexity. Multiplying $(-1)^{nm}b^{n-l}$ by $\mathrm{res}(g, r)$ requires just one multiplication in F. Since there are at most $m + 1$ recursive layers in total, this requires $O(m)$ arithmetic operations in F in total. Therefore the total cost of Algorithm 3.6 is $O(nm + m + n) = O(nm)$ arithmetic operations in F.

As with the Euclidean algorithm, it is beneficial to modify the algorithm so that the remainders are monic in order to avoid the coefficients growing large in size. We use

Lemma 3.38 (v) to implement the modification. E.g., in Step 6 of the algorithm, return $(-1)^{nm} b^{n-l} \operatorname{lc}(r)^m \operatorname{RES}(g, r/\operatorname{lc}(r))$. Making the next recursive call with monic r keeps the size of the coefficients of the remainders small.

One approach to improve the efficiency of the computation of a resultant is to adapt the FEEA, motivated by the fact the FEEA already computes some of the values that are needed to compute the resultant.

Given $f, g \in F[x]$, $\deg f, \deg g \geq 0$, let $\rho_0 = \operatorname{lc}(f)$, $\rho_1 = \operatorname{lc}(g)$, $r_0 = f/\rho_0$, $r_1 = g/\rho_1$, and consider the result of the monic EEA (Algorithm 3.4) applied to r_0 and r_1:

$$r_{i-1} = q_i r_i + \rho_{i+1} r_{i+1} \text{ for } 1 \leq i \leq l \text{ with } r_{l+1} = 0.$$

We define $d_i = \deg r_i$ for $0 \leq i \leq l$ and $d_{l+1} = -\infty$. ($\rho_{l+1} = \mathrm{u}(0) = 1$.) Consider the following theorem.

Theorem 3.40. *Given $f, g \in F[x]$, let l, r_i and ρ_i be as in the monic EEA as described above, so that $r_l = \gcd(f, g)$ with $d_l = \deg r_l$. Then*

$$\operatorname{res}(f, g) = \begin{cases} 0, & d_l > 0 \\ (-1)^\tau \rho_0^{d_1} \prod_{i=1}^{l} \rho_i^{d_{i-1}}, & d_l = 0, \end{cases}$$

where $\tau = \sum_{j=1}^{l-1} d_{j-1} d_j$.

Proof. If $d_l = \deg r_l > 0$, f and g have a common root and $\operatorname{res}(f, g) = 0$.

Now, suppose $d_l = 0$, i.e., $r_l = 1$. By Lemma 3.38 (v), we have

$$\operatorname{res}(f, g) = \operatorname{res}(\rho_0 r_0, \rho_1 r_1) = \rho_0^{d_1} \rho_1^{d_0} \operatorname{res}(r_0, r_1).$$

Now, $r_0 = q_1 r_1 + \rho_2 r_2$, so by Theorem 3.39, we have $\operatorname{res}(r_0, r_1) = (-1)^{d_0 d_1} \operatorname{res}(r_1, \rho_2 r_2)$. Then

$$\begin{aligned} \operatorname{res}(f, g) &= (-1)^{d_0 d_1} \rho_0^{d_1} \rho_1^{d_0} \operatorname{res}(r_1, \rho_2 r_2) \\ &= (-1)^{d_0 d_1} \rho_0^{d_1} \rho_1^{d_0} \rho_2^{d_1} \operatorname{res}(r_1, r_2). \end{aligned}$$

Next, from $r_1 = q_2 r_2 + \rho_3 r_3$, we get

$$\begin{aligned} \operatorname{res}(f, g) &= (-1)^{d_0 d_1} \rho_0^{d_1} \rho_1^{d_0} \rho_2^{d_1} \left[(-1)^{d_1 d_2} \rho_3^{d_2} \operatorname{res}(r_2, r_3) \right] \\ &= (-1)^{(d_0 d_1 + d_1 d_2)} \rho_0^{d_1} \rho_1^{d_0} \rho_2^{d_1} \rho_3^{d_2} \operatorname{res}(r_2, r_3). \end{aligned}$$

Continuing, we have

$$\operatorname{res}(f,g) = (-1)^{(d_0d_1+\cdots+d_{l-2}d_{l-1})}\rho_0^{d_1}\rho_1^{d_0}\rho_2^{d_1}\cdots\rho_l^{d_{l-1}}\operatorname{res}(r_{l-1},r_l).$$

By Lemma 3.38 (i), we have $\operatorname{res}(r_{l-1},r_l) = 1^{d_{l-1}} = 1$. Therefore

$$\operatorname{res}(f,g) = (-1)^{(d_0d_1+\cdots+d_{l-2}d_{l-1})}\rho_0^{d_1}\rho_1^{d_0}\rho_2^{d_1}\cdots\rho_l^{d_{l-1}},$$

as claimed. □

Theorem 3.40 indicates that if $\gcd(f,g) = 1$ and therefore $\operatorname{res}(f,g) \neq 0$, we can compute the resultant of f and g by determining the leading coefficients $\rho_2,\ldots,\rho_l$ of the remainders $r_2,\ldots,r_l$ and the respective degrees $d_2,\ldots,d_l$. (We already know $\rho_0 = \mathrm{u}(f), \rho_1 = \mathrm{u}(g), d_0 = \deg f$, and $d_1 = \deg g$.) Conveniently, the FEEA already computes ρ_i, $2 \le i \le l$, which are necessary to compute the matrices $Q_i = \begin{pmatrix} 0 & 1 \\ \rho_{i+1}^{-1} & -q_i\rho_{i+1}^{-1} \end{pmatrix}$, $i = 1,\ldots,l$. (Recall that we defined $\rho_{l+1} = 1$ for convenience.) Thus, if we can extract ρ_i from the FEEA instance and determine d_i, then we can compute the resultant at the cost of $O(\sum_{i=0}^{l}\log d_i) \in O(n\log n)$ arithmetic operations in F in addition to that of the FEEA.

The obvious way to obtain ρ_i is to simply store the leading coefficients as they are computed in Step 7 of Algorithm 3.5. However, we run into a problem with this approach, as some leading coefficients may be incorrect: Recall that in Step 4, the algorithm recursively obtains $j-1 = \eta^*(\lfloor\frac{\deg f}{2}\rfloor)$ and $R^* = Q_{j-1}^*Q_{j-2}\cdots Q_1$, where $Q_{j-1}^* = \begin{pmatrix} 0 & 1 \\ \rho_j^{*-1} & -q_{j-1}\rho_j^{*-1} \end{pmatrix}$ and $\begin{pmatrix} r_{j-1}^* \\ r_j^* \end{pmatrix} = R^*\begin{pmatrix} r_0^* \\ r_1^* \end{pmatrix}$. We would end up collecting $\rho_2,\rho_3,\ldots,\rho_{j-2},\rho_{j-1}^*$ from this recursive call. The solution is to multiply the stored ρ_j^* by $\tilde{\rho}_j$, which is computed in Step 5 to compute R from R^*, and store the new value $\rho_j^*\tilde{\rho}_j$ instead. This is indeed the correct ρ_j we want: in the proof of correctness of Algorithm 3.5, we showed that $\tilde{\rho}_j = \rho_j^{*-1}\rho_j$, so $\rho_j^* \times \tilde{\rho}_j = \rho_j$. We apply a similar modification to ρ_h^* after Step 12 to obtain ρ_h.

Another challenge in adapting the FEEA to compute resultants is that we are unable to collect the degrees of the remainders during the FEEA instance: The FEEA does not compute all the r_i in the remainder sequence for r_0 and r_1. Instead, the algorithm uses truncated polynomials for recursive calls to find q_i and ρ_i, and this may alter the degrees of the subsequent remainders. E.g., we may encounter $r_l^* = x^2$ when $r_l = 1$. As a result, we must determine d_i for $2 \le i \le l$. One useful fact is that $r_l = 1$ and $d_l = 0$, because we

already know that if $\gcd(f, g) \neq 1$, then $\operatorname{res}(f, g) = 0$. Our idea is to recover the d_i from the $\deg q_i$ and the knowledge that $d_l = 0$ once q_i are known, i.e., after the FEEA has completed. To determine the d_i, we need the following lemma.

Lemma 3.41. *$d_{i-1} = \deg q_i + d_i$ for $1 \leq i \leq l$.*

Proof. For $1 \leq i \leq l$, $\rho_{i+1} r_{i+1} = r_{i-1}$ rem $r_i = r_{i-1} - q_i r_i$ with $d_{i+1} < d_i < d_{i-1}$. It follows that $d_{i-1} = \deg r_{i-1} = \deg(q_i r_i) = \deg q_i + \deg r_i = \deg q_i + d_i$. □

Using Lemma 3.41 and the fact that $\gcd(f, g) = 1$ and $d_l = 0$, we can now compute $d_{l-1} = \deg q_l, d_{l-2} = \deg q_{l-1} + d_{l-1}, \ldots, d_2 = \deg q_3 + d_3$, in order, if we save the degrees of the quotients while the FEEA is running. Once we have retrieved all ρ_i and d_i, the resultant can be easily computed using the formula given in Theorem 3.40.

Theorem 3.42. *The cost of computing the resultant using the FEEA is $O(M(n) \log n)$.*

Proof. As discussed earlier, computing all ρ_i costs $O(n \log n)$ arithmetic operations in F. Computing d_i can be done quickly in l integer additions and does not contribute significantly to the overall cost of the algorithm. Recall that the FEEA requires $O(M(n) \log n)$ arithmetic operations in F. Since all multiplication algorithms cost more than $O(n)$, $n \in O(M(n))$ and $O(n \log n) \in O(M(n) \log n)$. Therefore the overall cost of computing the resultant using the FEEA is $O(n \log n + M(n) \log n) \in O(M(n) \log n)$. □

Remark 3.43. Given $f, g \in F[x]$, we let $\rho_0 = \operatorname{lc}(f), r_0 = f/\rho_0, \rho_1 = \operatorname{lc}(g)$, and $r_1 = g/\rho_1$ and use the monic polynomials r_0 and r_1, along with the integer $k = \deg f$, as inputs for the FEEA. The algorithm requires $\deg f = \deg r_0 > \deg r_1 = \deg g$, so some pre-adjustments may be required if $\deg f \leq \deg g$. Following are the necessary adjustments for the resultant computation. (See Remarks 3.34 for additional adjustments.)

Step 1a. if $\deg f < \deg g$: switch f and g and multiply the resultant returned by $(-1)^{nm}$.

Step 1b. if $\deg f = \deg g$ and $f/\operatorname{lc}(f) = g/\operatorname{lc}(g)$: return 1 if $\deg f = 0$ and 0 otherwise.

Step 1c. if $\deg f = \deg g$ and $f/\operatorname{lc}(f) \neq g/\operatorname{lc}(g)$: do a single division to compute $r = f$ rem g and call the algorithm to obtain $R = \operatorname{res}(g, r)$. Then we find

$$\operatorname{res}(f, g) = (-1)^{\deg f \deg g} \operatorname{lc}(g)^{\deg f - \deg r} R.$$

Chapter 4

Summary

In this thesis, we presented efficient algorithms for polynomial manipulation. In Chapter 2, we introduced a new algorithm to interpolate sparse polynomials over a finite field using discrete logarithms. Our algorithm is based on Ben-Or and Tiwari's deterministic algorithm for sparse polynomials with integer coefficients. We work over $\mathbb{Z}_p$, where p is a smooth prime of our choice, and the target polynomial is represented by a black box. We compared the new algorithm against Zippel's probabilistic sparse interpolation algorithm and showed the timings from implementations of both. The benchmarks showed that our algorithm performs better than Zippel's algorithm for sparse polynomials, benefitting from fewer probes to the black box and the fact the new algorithm does not interpolate one variable at a time. However, Zippel's algorithm proved to be more efficient for dense polynomials.

To interpolate a polynomial with t nonzero terms, we need to compute the roots of $\Lambda(z) = z^t + \lambda_{t-1} z^{t-1} + \cdots + \lambda_0$. We used Rabin's root-finding algorithm in our implementation of the interpolation algorithm, which computes a series of polynomial GCDs. Thus we saw that the ability to efficiently compute polynomial GCDs is critical for our interpolation algorithm, and for large t, say $t \geq 10^6$, the Euclidean algorithm in which lies complexity $O(t^2)$ is too slow. Motivated by this observation, we reviewed in Chapter 3 the Fast Extended Euclidean algorithm (FEEA), which divides the division steps of the Euclidean algorithm into two recursive tasks of roughly the equal sizes and a single division. We implemented the classical and fast versions of the Euclidean algorithm and presented the respective benchmarks. While the FEEA was not as fast as the traditional Euclidean algorithm for polynomials of relatively low degrees due to the heavy overhead cost associated with matrix operations, the results were quickly reversed for polynomials of high degree to demonstrate

the clear advantage of using the FEEA over the traditional algorithm. We showed how the FEEA can be easily modified to compute the resultant of two polynomials at not much additional cost.

Bibliography

[1] Bareiss, E. H. Sylvesters identity and multistep integer-preserving gaussian elimination. *Mathematics of Computation 22* (1968), 565 – 578.

[2] Ben-Or, M. Probabilistic algorithms in finite fields. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science* (Washington, DC, 1981), FOCS '81, IEEE Computer Society, pp. 394–398.

[3] Ben-Or, M., and Tiwari, P. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing* (New York, 1988), STOC '88, ACM Press, pp. 301–309.

[4] Berlekamp, E. R. *Algebraic Coding Theory.* McGraw-Hill, New York, 1968.

[5] Bläser, M., Hardt, M., Lipton, R. J., and Vishnoi, N. K. Deterministically testing sparse polynomial identities of unbounded degree. *Information Processing Letters 109*, 3 (2009), 187–192.

[6] Bostan, A., Salvy, B., and Éric Schost. Fast algorithms for zero-dimensional polynomial systems using duality. In *Applicable Algebra in Engineering, Communication and Computing* (2001), pp. 239–272.

[7] Brown, W. S. On Euclid's algorithm and the computation of polynomial greatest common divisors. *Journal of the ACM 18*, 4 (1971), 478–504.

[8] Cojocaru, A., Bruce, J. W., and Murty, R. *An Introduction to Sieve Methods and Their Applications.* London Mathematical Society Student Texts. Cambridge University Press, 2005.

[9] Cox, D. A., Little, J., and O'Shea, D. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*, third ed. Springer, 2007.

[10] Fine, B., and Rosenberger, G. *Number Theory: An Introduction via the Distribution of Primes.* Birkhäuser Boston, 2006.

[11] Gathen, J. v. z., and Gerhard, J. *Modern Computer Algebra*, second ed. Cambridge University Press, New York, 2003.

[12] Geddes, K. O., Czapor, S. R., and Labahn, G. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[13] Gentleman, W. M., and Johnson, S. C. Analysis of algorithms, a case study: Determinants of matrices with polynomial entries. *ACM Transactions on Mathematical Software 2*, 3 (1976), 232–241.

[14] Grabmeier, J., Kaltofen, E., and Weispfenning, V., Eds. *Computer Algebra Handbook: Foundations, Applications, Systems*. Springer, 2003.

[15] Hardy, G. H., and Wright, E. M. *An Introduction to the Theory of Numbers*, fifth ed. Oxford University Press, 1979.

[16] Javadi, S. M. M. *Efficient Algorithms for Computations with Sparse Polynomials*. PhD thesis, Simon Fraser University, 2010.

[17] Javadi, S. M. M., and Monagan, M. Parallel sparse polynomial interpolation over finite fields. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation* (New York, 2010), PASCO '10, ACM Press, pp. 160–168.

[18] Kaltofen, E., and Lakshman, Y. N. Improved sparse multivariate polynomial interpolation algorithms. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation* (London, 1989), ISSAC '88, Springer-Verlag, pp. 467–474.

[19] Kaltofen, E., Lakshman, Y. N., and Wiley, J.-M. Modular rational sparse multivariate polynomial interpolation. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation* (New York, 1990), ISSAC '90, ACM Press, pp. 135–139.

[20] Kaltofen, E., Lee, W.-s., and Lobo, A. A. Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation* (New York, 2000), ISSAC '00, ACM Press, pp. 192–201.

[21] Kaltofen, E., and Trager, B. M. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *Journal of Symbolic Computation 9*, 3 (1990), 301–320.

[22] Khodadad, S. Fast rational function reconstruction. Master's thesis, Simon Fraser University, 2005.

[23] Knuth, D. The analysis of algorithms. *In Actes du Congrès International des Mathématiciens 3* (1970), 269–274.

[24] Lehmer, D. Euclid's algorithm for large numbers. *The American Mathematical Monthly 45*, 4 (1938), 227–233.

[25] Massey, J. L. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory 15* (1969), 122–127.

[26] Mignotte, M. *Mathematics for Computer Algebra.* Springer-Verlag, 1992.

[27] Moenck, R. T. Fast computation of GCDs. In *Proceedings of the 5th annual ACM Symposium on Theory of Computing* (New York, 1973), STOC '73, ACM Press, pp. 142–151.

[28] Pohlig, S., and Hellman, M. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory 24* (1978), 106–110.

[29] Rabin, M. O. Probabilistic algorithms in finite fields. *SIAM Journal on Computing 9* (1980), 273–280.

[30] Schönhage, A. Schnelle berechnung von kettenbruchentwicklungen. *Acta Informatica 1* (1971), 139–144.

[31] Schwartz, J. T. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM 27*, 4 (1980), 701–717.

[32] Shanks, D. Class number, a theory of factorization and genera. In *Proceedings of Symposia in Pure Mathematics* (1971), vol. 20, AMS, pp. 415–440.

[33] Stinson, D. *Cryptography: Theory and Practice, Second Edition.* CRC/Chapman & Hall, 2002.

[34] Zippel, R. E. Probabilistic algorithms for sparse polynomials. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation* (London, 1979), EUROSAM '79, Springer-Verlag, pp. 216–226.

[35] Zippel, R. E. Interpolating polynomials from their values. *Journal of Symbolic Computation 9*, 3 (1990), 375–403.

[36] Zippel, R. E. *Effective Polynomial Computation.* Kluwer Academic Publishers, 1993.