FAST RATIONAL FUNCTION RECONSTRUCTION

by

Sara Khodadad
B.Sc., Sharif University of Technology, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Sara Khodadad 2005
SIMON FRASER UNIVERSITY
November 2005

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Sara Khodadad

Degree: Master of Science

Title of thesis: Fast Rational Function Reconstruction

Examining Committee: Dr. Daniel Weiskopf
Chair

____

Dr. Michael Monagan, Senior Supervisor

____

Dr. Arvind Gupta, Supervisor

____

Dr. Petr Lisonek, Examiner

Date Approved: ______________________________

Abstract

Let F be a field, $f, g \in F[x]$ with $m = \deg f > \deg g \geq 0$. Our problem is to find a rational function $n/d \in F(x)$ where $n/d \equiv g \bmod f$, $\gcd(f, d) = \gcd(n, d) = 1$ and $\deg n + \deg d < m$. If degree bounds $N \geq \deg n$ and $D \geq \deg d$ satisfying $N + D < m$ are known, then the problem is solved by the Extended Euclidean Algorithm in $F[x]$. If degree bounds are not known it is still possible to find n/d with high probability. One way is to use maximal quotient rational function reconstruction. We have implemented the algorithm for $F[x] = \mathbb{Z}_p[x]$, with p a prime. To speed up the algorithm, our implementation uses Karatsuba's algorithm for multiplication in $\mathbb{Z}_p[x]$ and a Fast Extended Euclidean Algorithm. As an application, we have modified Brown's modular GCD algorithm to use the maximal quotient algorithm. The modification reduces the number of evaluation points needed by the algorithm.

Contents

Approval ii

Abstract iii

Contents iv

List of Tables vi

List of Figures vii

List of Algorithms viii

1 Introduction 1

2 Fast Polynomial Arithmetic 6

2.1 Fast Polynomial Multiplication 6

2.1.1 Karatsuba's Multiplication Algorithm 7

2.1.2 Memory Requirements of Karatsuba's Algorithm 12

2.2 The Euclidean Algorithm 14

2.2.1 The Extended Euclidean Algorithm 15

2.2.2 The Fast Extended Euclidean Algorithm 19

2.2.3 Fast Polynomial Interpolation (Application) 30

3 Rational Function Reconstruction 32

3.1 Rational Function Interpolation (Cauchy Interpolation) 32

3.2 Rational Function Reconstruction (RFR) 34

3.2.1 Wang's Algorithm 35

3.2.2 Maximal Quotient Rational Function Reconstruction 37

4 Polynomial GCD Computation **45**

4.1 Multivariate GCD Computation (Brown's Algorithm) 45

4.2 Application of RFR to Brown's Algorithm 50

5 Summary **54**

Bibliography **55**

List of Tables

2.1 The classical and Karatsuba's multiplication algorithm timings (in ms) 11
2.2 The number of multiplications and additions of steps of Algorithm 2.5 28
2.3 Timings (in ms) of the FEEA compared to the EEA 30

List of Figures

2.1 Steps of Karatsuba's multiplication algorithm 9
2.2 Timings (in ms) of Karatsuba's algorithm for different cutoff degrees 12
2.3 Memory requirements of Karatsuba's algorithm in our implementation 13
2.4 Timings (in ms) of the FEEA for different cutoff degree 29

List of Algorithms

2.1 Karatsuba's Algorithm for input polynomials of size $n = 2^k$ 8
2.2 Karatsuba's Algorithm for polynomials of different sizes 10
2.3 Classical Euclidean Algorithm (EA) 16
2.4 Extended Euclidean Algorithm (EEA) 16
2.5 Fast Extended Euclidean Algorithm (FEEA) 23
3.1 Wang's Rational Function Reconstruction Algorithm 36
3.2 Modified FEEA to return the maximal quotient(MQFEEA) 40
3.3 Maximal Quotient Rational Function Reconstruction (MQRFR) 43
4.1 Brown's multivariate reduction algorithm (PGCD) 49
4.2 Retrieval of the leading coefficient (LCR) 51

Chapter 1

Introduction

Rational reconstruction has become an important tool with many applications in computer algebra. It enables the algorithms to recover rational numbers from their images modulo a large integer (a prime, a prime power or product of several primes) or to recover rational functions from their images modulo a given polynomial.

Let F be a field. Given a rational function $n/d \in F(x)$ and a polynomial $m \in F[x]$ where $\deg m > 0$ and $\gcd(m, d) = \gcd(n, d) = 1$, we can easily compute $u \in F[x]$ such that $u \equiv n/d \mod m$. The rational function reconstruction algorithm presents a solution for the reverse problem. That is, for given polynomials $m, u \in F[x]$ where $0 \le \deg u < \deg m$ it outputs a rational function $n/d \in F(x)$ where $n/d \equiv u \mod m$ and $\gcd(d, m) = \gcd(n, d) = 1$.

The rational function reconstruction problem does not necessarily have a unique solution. The Extended Euclidean Algorithm finds all solutions satisfying $\deg n + \deg d < \deg m$. However, it is not hard to see that there is only one solution when degree bounds $N \ge \deg n$ and $D \ge \deg d$ satisfying $N + D < \deg m$ are given.

Example 1.1. Let $F = \mathbb{Z}_7$. We are given

$$f(1) = 5, f(2) = 2, f(3) = 1,$$

where $f \in \mathbb{Z}_7[x]$. We want to find a rational function $n/d \in \mathbb{Z}_7(x)$ such that

$$\frac{n(\alpha)}{d(\alpha)} = f(\alpha), \quad d(\alpha) \neq 0, \quad \alpha \in \{1, 2, 3\}.$$

Using polynomial interpolation we can easily compute $u = x^2 + x + 3$ satisfying $u(\alpha) = f(\alpha)$, and rewrite the above problem in the form of the following rational function reconstruction

problem:

Given $m, u \in \mathbb{Z}_7[x]$ with $m = (x-1)(x-2)(x-3)$ and $u = x^2 + x + 3$, find a rational function $n/d \in \mathbb{Z}_7(x)$ such that

$$n/d \equiv u \mod m, \quad \gcd(m, d) = 1.$$

Let $N = 1$ and $D = 1$ be respectively degree bounds for the numerator and the denominator of the solution. Using the Extended Euclidean Algorithm we get the following solutions

$$\frac{n_1}{d_1} = \frac{x^2+x+3}{1}, \quad \frac{n_2}{d_2} = \frac{6x+6}{x}, \quad \frac{n_3}{d_3} = \frac{3}{x^2+1}.$$

Among these 3 solutions only n_2/d_2 satisfies the degree bound requirement, that is, $\deg n_2 \leq 1$ and $\deg d_2 \leq 1$.

In case degree bounds N, D are not available we can use either Wang's algorithm (Algorithm 3.1) or the maximal quotient rational reconstruction algorithm (Algorithm 3.3). Both of these algorithms require an external mechanism that enables us to check whether or not the output of the algorithm is the one we were expecting. In this example we assume that this mechanism gives us $u(\alpha_i)$ with $\alpha_i \in \mathbb{Z}_7$ a new evaluation point.

The output of Wang's algorithm with inputs $m = (x-1)(x-2)(x-3), u = x^2 + x + 3$ would be $(6x+6)/x$. Assuming $u(4) = 5$ we have $m = (x-1)(x-2)(x-3)(x-4)$ and $u = 4x^3 + 5x^2 + 3x$. This time Wang's algorithm returns $(x^2+2)/(x+1)$. Adding another point $(5, 1)$ or $u(5) = 1$ and calling Wang's algorithm with inputs $m = (x-1)(x-2)(x-3)(x-4)(x-5)$ and $u = 4x^4 + 6x^3 + 5x^2 + 6x + 5$ we get the same solution. At this point we require an external mechanism to check whether $(x^2+1)/(x+1)$ is the expected solution or not.

Background

In 1981, Wang [1] presented a new algorithm for the partial fraction decomposition of rational functions in $\mathbb{Q}(x)$. To get a more efficient algorithm he suggested that one first solve the problem modulo a suitably selected prime and then lift the problem p-adically to get the desired solution over $\mathbb{Q}(x)$. When devising this algorithm he encountered the rational number reconstruction problem and developed an algorithm that enabled him to reconstruct rational coefficients of a polynomial in $\mathbb{Q}[x]$ from their images modulo $m = p^k$,

a prime power. The algorithm gets $m, u \in \mathbb{Z}$ as input and outputs a rational number n/d satisfying $n/d \equiv u \mod m$ if a solution exists.

Wang showed, by adding the requirement $0 \leq |n|, d < \sqrt{m/2}$, or equivalently $m > 2(\max(|n|, d))^2$, that the algorithm uniquely determines the solution if it exists. In fact, Wang's algorithm is the Extended Euclidean Algorithm equipped with a different stopping condition. Wang did not provide any proof as to the correctness of his algorithm in the original paper, however in [2] Wang, Guy and Davenport proved that if there is a solution, it will be found by his algorithm.

Since then, Wang's algorithm has been used in many contexts including polynomial factorization (see [3] and [4]), Gröbner basis computations over $\mathbb{Q}$ (see [5] and [6]), polynomial interpolation (see [7]), solving linear systems over $\mathbb{Q}$ (see [8]) and polynomial GCD computation (see [9] and [10]).

Monagan in [11] presented a more efficient solution for the rational reconstruction problem, which he called *Maximal Quotient Rational Reconstruction*. His algorithm also runs the Extended Euclidean algorithm on inputs m, u and outputs the rational number r_i/t_i where i represents the index of the maximal quotient q_i appearing in the Euclidean algorithm. He introduced input T to the algorithm and claimed that if we determine a good value for T such that $m > |n|dT$ then with high probability the algorithm outputs n/d for q the maximal quotient. He also stated that his algorithm can be applied to the rational function reconstruction problem over a finite field with p elements.

Wang's algorithm can be easily modified to solve the problem of rational function reconstruction as well. Von zur Gathen and Gerhard in [12, sec. 5.7] show how to use the Extended Euclidean Algorithm to solve the rational function reconstruction problem. The algorithm in addition to polynomials m and u, gets a third input $k \in \{0, \ldots, \deg m\}$ and outputs n, d such that $\deg n < k$ and $\deg d \leq \deg m - k$. The problem with this algorithm is that it needs the degree bound k for the numerator and hence, $\deg m - k$ for the denominator which is not always available in advance.

In this thesis we present a fast algorithm for the rational function reconstruction problem which is based on Monagan's maximal quotient algorithm. We show that if $\deg m > \deg n + \deg d + 1$, then with high probability the algorithm outputs n/d. The advantage of this algorithm is that it requires no degree bounds as in [12] and it requires approximately one more point than the minimum necessary i.e. $\deg n + \deg d + 1$ to reconstruct n/d.

Let $m_j = \prod_{i=1}^{j}(x - \alpha_i)$ where $\alpha_i \in F$ and $\deg m_j = \deg n + \deg d + 1$. To interpolate

n/d we need at least j points, hence, m_j is the smallest polynomial in terms of degree for which the rational function reconstruction can succeed in recovering n/d. Our algorithm has the following properties:

- for m_k with $k > j$ our algorithm outputs n/d with high probability, such that we need approximately one more evaluation point than the minimum necessary,
- if $\deg m_k > 2(\deg n + \deg d)$ then our algorithm outputs n/d with probability 1, and
- if $k \leq j$ the algorithm fails with high probability.

Similar to any rational reconstruction algorithm, our algorithm is based on the Extended Euclidean Algorithm. In order to be more efficient it uses the Fast Extended Euclidean Algorithm which has time complexity $O(\mathsf{M}(n)\log n)$, where $\mathsf{M}(n)$ denotes the number of field operations required to multiply two polynomials of degree less than n. For fast polynomial multiplication, we have implemented Karatsuba's algorithm which runs in $O(n^{\log_2 3})$.

To show one of the applications of our algorithm we have implemented an algorithm for computing the GCD of two multivariate polynomials. We have modified Brown's modular GCD algorithm to use the maximal quotient algorithm. This not only solves the leading coefficient problem in this algorithm but also reduces the number of trial division attempts to 1 with high probability. Moreover, the number of evaluation points required to recover the coefficients of the GCD is reduced.

Outline

In Chapter 2 we will explain three fast polynomial arithmetic operations, namely fast polynomial multiplication, the Fast Extended Euclidean Algorithm (FEEA) and fast polynomial interpolation. In order for the FEEA to be fast, one must implement the fast multiplication algorithm carefully. In Section 2.1 we discuss Karatsuba's multiplication algorithm and how it can be implemented so that it only uses a linear amount of memory with respect to the size of input polynomials. In Section 2.2 we explain the Extended Euclidean Algorithm and a fast version of it which is going to be the main part of the Rational Function Reconstruction algorithm. "Modern Computer Algebra" [12] is the main reference for the material discussed in this chapter.

In Chapter 3 we will explore the rational function reconstruction problem. First we

describe the problem and then we present Wang's algorithm. Next our algorithm is introduced. For a fast solution, we show how to modify the FEEA to do maximal quotient rational reconstruction.

Brown's modular algorithm for multivariate GCD computation is described in Chapter 4. We show how we can use rational function reconstruction presented in Chapter 3 to make Brown's algorithm work more efficiently. We show this modification reduces the number of evaluation points needed by the algorithm.

In the last chapter, we give a summary of what we have done in this thesis.

Chapter 2

Fast Polynomial Arithmetic

In this chapter we will mainly explore the Fast Extended Euclidean Algorithm. We refer to a polynomial's degree plus one as the size of the polynomial. For univariate polynomials of size at most n over a field, this algorithm finds all of the quotients and a single remainder r (which is the GCD) together with corresponding values of s and t, satisfying $as + bt = r$, using $O(\mathsf{M}(n) \log n)$ field operations. $\mathsf{M}(n)$ denotes the number of field operations required to multiply two univariate polynomials of size n.

In the first section of this chapter we will introduce Karatsuba's algorithm as a fast multiplication algorithm. In the second section we will present the Fast Extended Euclidean Algorithm and its application for solving the polynomial interpolation problem fast.

2.1 Fast Polynomial Multiplication

The polynomial multiplication algorithms which are asymptotically faster than the classical $O(n^2)$ method are considered to be fast polynomial multiplication algorithms. There are two well-known fast multiplication algorithms, namely Karatsuba's [13] algorithm and an FFT*-based multiplication algorithm. In the case of univariate polynomials of size n, the classical method uses $O(n^2)$ steps to compute the product, while Karatsuba's algorithm has a time complexity of $O(n^{1.58})$ and the FFT-based algorithm costs $O(n \log n)$. The FFT-based algorithm is asymptotically the fastest known algorithm for multiplication but it is more complicated to implement. Also Karatsuba's algorithm is faster than the FFT up to a

*Fast Fourier Transform

certain size, e.g. in MAGMA, Karatsuba's algorithm for integer multiplication is faster than the FFT for integers of size up to 50,000 bits.

2.1.1 Karatsuba's Multiplication Algorithm

Let R be a ring and $a, b \in R[x]$ of size n. The classical multiplication method uses n^2 multiplications and $(n-1)^2$ additions in R to compute ab. Yet, we can compute the product faster if we use Karatsuba's multiplication algorithm.

For simplicity assume that $n = 2^k$ for some $k \in \mathbb{N}$. Split a and b into two polynomials of size $n/2$:

$$a = a_2 x^{n/2} + a_1 \quad (a_1, a_2 \in R[x]) \tag{2.1}$$

$$b = b_2 x^{n/2} + b_1 \quad (b_1, b_2 \in R[x]). \tag{2.2}$$

Then the product ab can be written as

$$\begin{aligned} ab &= a_2 b_2 x^n + (a_1 b_2 + a_2 b_1) x^{n/2} + a_1 b_1 \\ &= a_2 b_2 x^n + ((a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2) x^{n/2} + a_1 b_1. \end{aligned} \tag{2.3}$$

Relation (2.3) can be used for computing $a_1 b_1$, $a_2 b_2$ and $(a_1 + a_2)(b_1 + b_2)$ recursively. This results in Algorithm 2.1.

Figure 2.1 illustrates Karatsuba's algorithm step by step. As shown in this figure, computing the product ab requires three multiplications and two additions on polynomials of size $n/2$, two subtractions on polynomials of size $n-1$ and one addition of size $n-2$.

Remark 2.1. Note that there is a "gap" between c_1 and $c_3 x^n$.

Let $T(n)$ denote the cost of multiplying two polynomials of size n. The following table shows the cost of each step of Algorithm 2.1 as illustrated in Figure 2.1.

Step	1	2	3	4	5
Cost	–	–	$3T(n/2) + n$	$2(n-1)$	$n-2$

ALGORITHM 2.1: Karatsuba's Algorithm for input polynomials of size $n = 2^k$

Input: Polynomials $a, b \in R[x]$ of size $n = 2^k$, where R is a ring and $\deg a = \deg b = n - 1$.
Output: Polynomial $c = ab \in R[x]$.

1. if $n = 1$ then return $a \cdot b \in R$
2. let $a = a_2 x^{n/2} + a_1$ and $b = b_2 x^{n/2} + b_1$ where $a_1, a_2, b_1, b_2 \in R[x]$ are of size $n/2$
3. compute $c_1 = a_1 b_1$, $c' = (a_1 + a_2)(b_1 + b_2)$ and $c_3 = a_2 b_2$ by recursively applying the algorithm
4. compute $c_2 = c' - c_1 - c_3$
5. return $c = c_3 x^n + c_2 x^{n/2} + c_1$

Assuming $T(1) = 1$ we have the following recurrence relation:

$$\begin{aligned}
T(n) &= 3T(n/2) + 4n - 4 \\
&= 3^2 T(n/2^2) + 3(4(n/2) - 4) + 4n - 4 \\
&\vdots \\
&= 3^k T(n/2^k) + 4 \sum_{i=0}^{k-1} ((3/2)^i n - 3^i) \\
&= 3^k T(1) + 8n((3/2)^k - 1) - 2(3^k - 1) \\
&= 7n^{\log_2 3} - 8n + 2.
\end{aligned}$$

Hence

$$T(n) \in O(n^{\log_2 3}) = O(n^{1.585}).$$

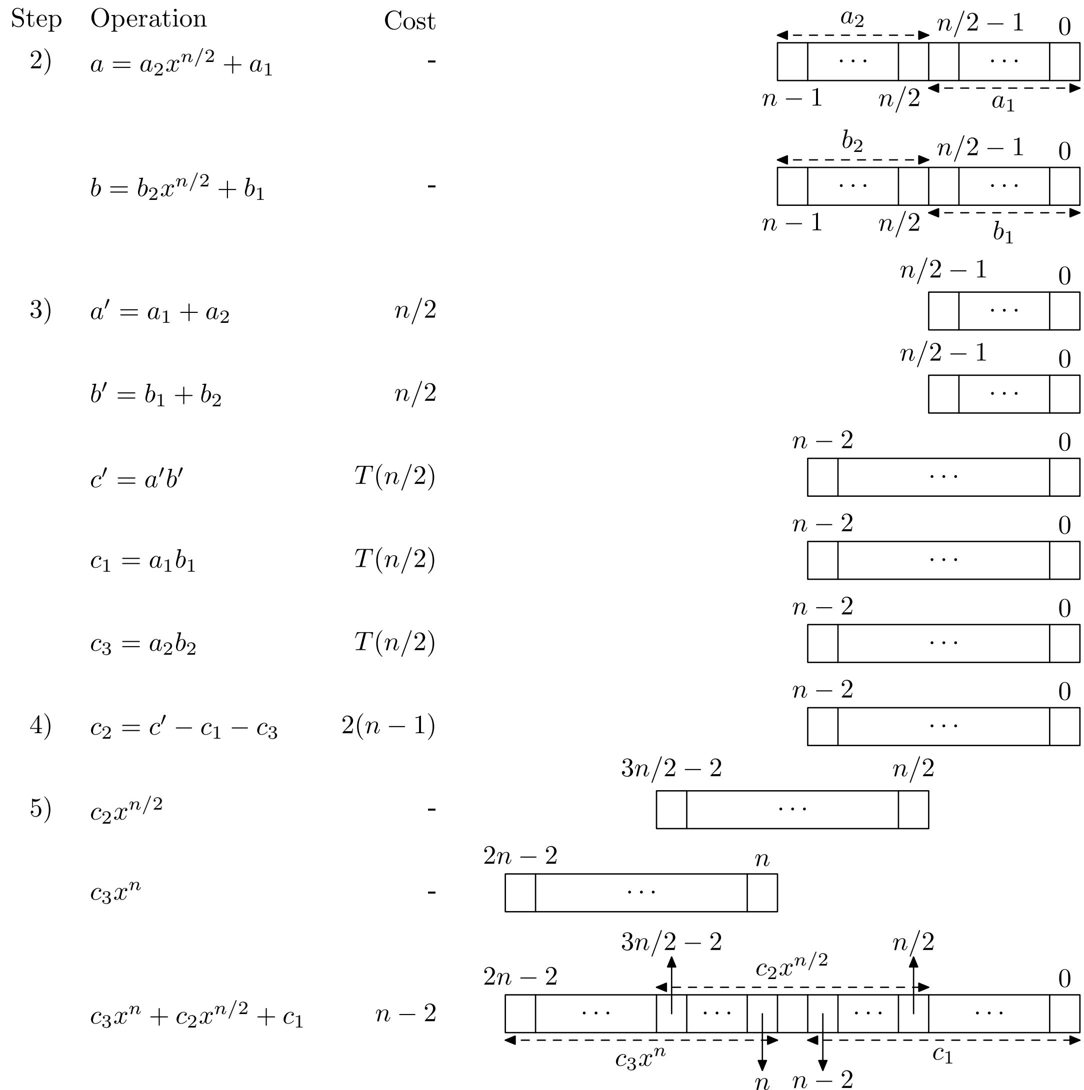

Figure 2.1: Steps of Karatsuba's multiplication algorithm

In case n is not a power of 2 but a and b are still of equal size, we can assume that they are both of size $2^{\lceil \log_2 n \rceil}$ with some top coefficients equal to zero. However, this may not be efficient if n is only slightly larger than a power of 2. Alternately, a and b can be divided into sub-polynomials of size $n_1 = \lceil n/2 \rceil$ and $n_2 = \lfloor n/2 \rfloor$. It is obvious that if n is even then $n_1 = n_2 = n/2$ and otherwise $n_1 = n_2 + 1$. Assuming we split the polynomials in a way that the lower half is of size n_1 and the upper half is of size n_2, we will have

$$a = a_2 x^{n_1} + a_1 \quad (a_1, a_2 \in R[x])$$
$$b = b_2 x^{n_1} + b_1 \quad (b_1, b_2 \in R[x])$$

and

$$\begin{aligned} ab &= a_2 b_2 x^{2n_1} + (a_1 b_2 + a_2 b_1) x^{n_1} + a_1 b_1 \\ &= a_2 b_2 x^{2n_1} + ((a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2) x^{n_1} + a_1 b_1. \qquad (2.4) \end{aligned}$$

Algorithm 2.1 can easily be modified to compute the product of two input polynomials of the same size, not necessarily a power of 2, without affecting the asymptotic complexity.

We next consider the case where input polynomials are not of the same size. Algorithm 2.2 describes Karatsuba's multiplication algorithm in this case.

Algorithm 2.2: Karatsuba's Algorithm for polynomials of different sizes

---

Input: Polynomials $a, b \in R[x]$ where R is a ring, $m = \deg b + 1$, $n = \deg a + 1 = qm + r$ ($r < m$) and $n \geq m$
Output: Polynomial $c = ab \in R[x]$.

1. let $a = a_q x^{mq} + a_{q-1} x^{m(q-1)} + \ldots + a_1 x^m + a_0$, where all a_i's are in $R[x]$ and of size m except a_q which is of size r

2. compute $c_i = a_i b$ for $0 \leq i < q$ using Algorithm 2.1 (after a small modification), and $c_q = a_q b$ by recursively calling Algorithm 2.2.

3. return $c = c_q x^{mq} + c_{q-1} x^{m(q-1)} + \ldots + c_1 x^m + c_0$

Let a and b be two polynomials of size n and m respectively. Without loss of generality assume $n \geq m$. Let q and r be respectively the quotient and the remainder of dividing n by

m, i.e. $n = qm + r$ ($r < m$). In the first step of Algorithm 2.2 polynomial a is divided into chunks of size at most m namely a_is as follows

$$a = a_q x^{mq} + a_{q-1} x^{m(q-1)} + \ldots + a_1 x^m + a_0,$$

so we will have

$$ab = (a_q b) x^{mq} + (a_{q-1} b) x^{m(q-1)} + \ldots + (a_1 b) x^m + (a_0 b).$$

Polynomial b and all a_i's except a_q are of size m. Thus, Algorithm 2.1 can be applied for computing $c_i = a_i b$ ($0 \le i < q$). But a_q and b are of different sizes so to compute $c_q = a_q b$ we recursively use Algorithm 2.2 with b and a_q as inputs. In the last step of Algorithm 2.2 we perform q additions and obtain the product

$$c = c_q x^{mq} + c_{q-1} x^{m(q-1)} + \ldots + c_1 x^m + c_0.$$

In practice for small input polynomials the classical method performs better than Karatsuba's algorithm. Therefore, a hybrid implementation which makes use of both algorithms is the best choice. We find a *cutoff* degree above which we use Karatsuba's algorithm and below that the classical method is applied. The cutoff degree can be computed by running both algorithms on random input polynomials of increasing size. We can easily incorporate this change in Algorithms 2.1 and 2.2.

Figure 2.2 shows the timings of Karatsuba's algorithm (hybrid implementation) on two random polynomials of degree 1500 with the cutoff degree changing from 10 to 100. As illustrated, the best cutoff degree is 55.

n	Karatsuba	Classical
128	0.34	0.38
256	0.98	1.40
512	2.93	5.40
1024	8.93	21.62
2048	26.48	84.43
4096	79.78	345.67
8192	245.04	1375.42

Table 2.1: The classical and Karatsuba's multiplication algorithm timings (in ms)

The data in Table 2.1 includes the timings, in milliseconds, we gathered for our implementation of Karatsuba's algorithm and the classical multiplication method (in Java) over

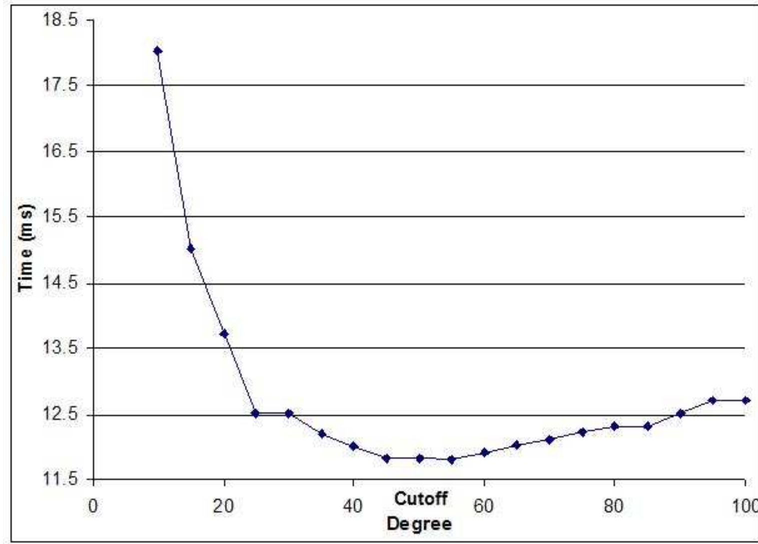

Figure 2.2: Timings (in ms) of Karatsuba's algorithm for different cutoff degrees

$\mathbb{Z}_p[x]$, where n denotes the polynomial degree and p is a 15 bit prime. The timings for Karatsuba's algorithm increase by a factor close to 3 as the degree doubles which confirms that our implementation of Karatsuba's algorithm is of time complexity $O(n^{\log_2 3})$.

2.1.2 Memory Requirements of Karatsuba's Algorithm

A naive implementation of Karatsuba's algorithm makes use of some extra storage in each recursive call to the algorithm. Let $M(n)$ denote the total amount of memory required to multiply polynomials a and b of size n using Karatsuba's algorithm, where $n = 2^k$ for some $k \in \mathbb{N}$. The following table displays the amount of memory used in each step of Algorithm 2.1.

Step	1	2	3	4	5
Memory	–	$2n$	$3M(n/2) + n$	$n - 1$	$2n - 1$

Note that $M(n)$ does not include the memory required to store a and b which is itself $2n$ (step 2). Assuming $M(1) = 1$ we will have

$$\begin{aligned} M(n) &= n/2 + n/2 + 3M(n/2) + n - 1 + 2n - 1 \\ &= 8n^{\log_2 3} - 8n + 1 \in O(n^{\log_2 3}). \end{aligned}$$

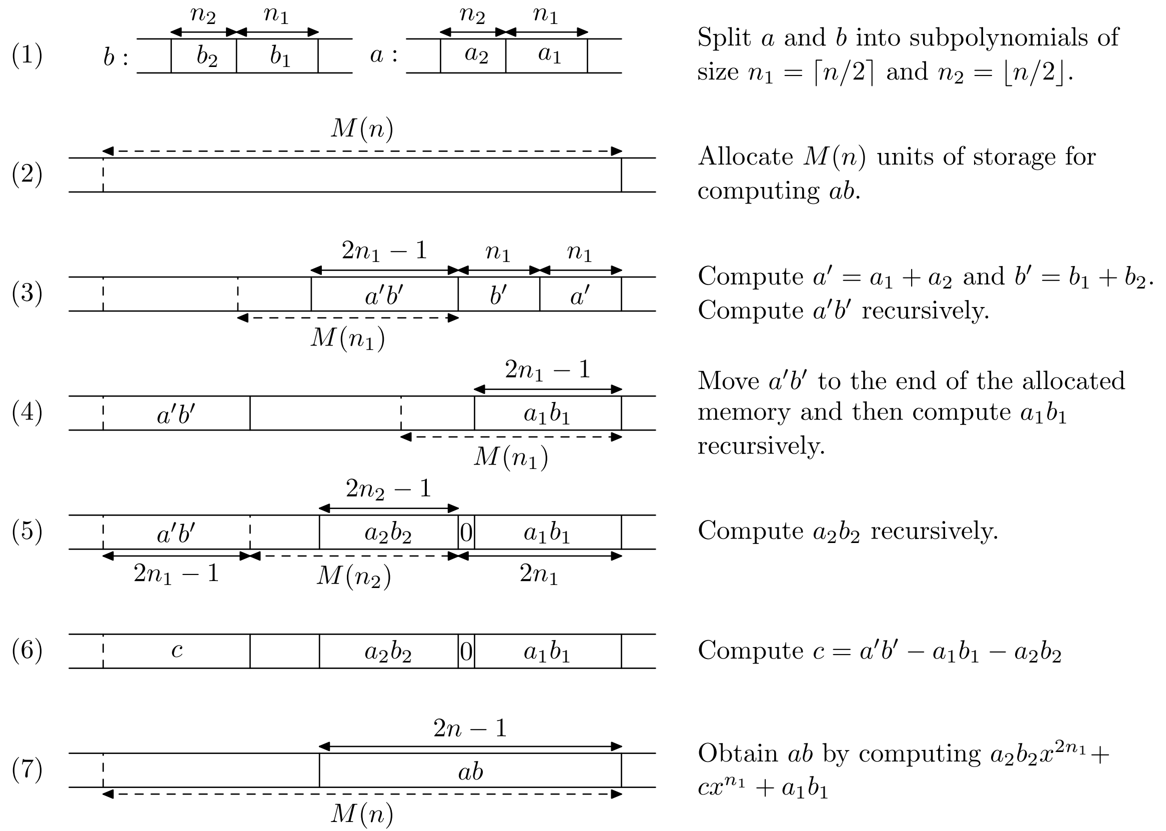

Figure 2.3: Memory requirements of Karatsuba's algorithm in our implementation

Therefore, the total amount of memory required to compute ab using Karatsuba's algorithm is of order $O(n^{\log_2 3})$. In 1993 Maeder in [14] suggested an "in place" implementation for Karatsuba's integer multiplication algorithm. He gave upper and lower bounds for the amount of auxiliary storage required.

We used the same method in our implementation of Karatsuba's algorithm for polynomial multiplication. In our implementation the total amount of memory required for multiplying the input polynomials—taking into account the memory required for performing intermediate calculations—is computed and allocated in advance, and passed as a parameter to the method implementing the multiplication algorithm.

Let a and b be two polynomials of the same size which is not necessarily a power of 2. Figure 2.3 illustrates the order in which the computations must be done so that the number

of necessary copies is minimized and the results are put in their final location. We show below that by implementing the algorithm "in place" only $4n$ words of memory are required which is linear in the size of the input polynomials.

In Figure 2.3 the total amount of memory required for any multiplication is marked by dashed lines and arrows, while the required amount of memory to keep the result only, is marked by solid lines. Row (5) shows that it is sufficient to allocate totally $2n_1 + M(n_2) + 2n_1 - 1$ memory cells for computing ab, in other words

$$M(n) = \begin{cases} M(\lfloor n/2 \rfloor) + 4\lceil n/2 \rceil - 1, & n \geq 3; \\ 3, & n = 2; \\ 1, & n = 1. \end{cases} \tag{2.5}$$

We claim $4n$ is an upper bound for $M(n)$ and prove our claim by strong induction. So we must show that the following inequality holds for all integer values of n:

$$M(n) < 4n. \tag{2.6}$$

The basis is to verify that $M(1) = 1 < 4$ and $M(2) = 3 < 8$. Now we must prove that if (2.6) holds for all $k \leq n-1$, it also holds for $k = n$,

$$M(n) = M(n_2) + 4n_1 - 1 < 4n_2 + 4n_1 - 1 = 4n - 1 < 4n. \quad \square$$

2.2 The Euclidean Algorithm

The Euclidean algorithm finds the greatest common divisor of two integers or two polynomials. However, it has a number of nice properties and applications which go far beyond that of just computing greatest common divisors.

In Section 2.2.1 we describe how the Classical and the Extended Euclidean Algorithms work and investigate some properties of the latter algorithm. Then in Section 2.2.2 we will explore the Fast Extended Euclidean Algorithm, also called Half-GCD. Given two polynomials of size n with coefficients from a field F, the Extended Euclidean Algorithm uses $O(n^2)$ field operations to compute their greatest common divisor. However, the fast Euclidean algorithm computes the same GCD in $O(\mathsf{M}(n) \log n)$ field operations, where $\mathsf{M}(n)$ denotes the number of field operations required to multiply two univariate polynomials of size n. Hence using Karatsuba's multiplication the GCD can be computed using $O(n^{\log_2 3} \log n)$ field operations.

2.2.1 The Extended Euclidean Algorithm

In Chapter 3 we will introduce an algorithm for reconstructing a rational function from its image modulo a univariate polynomial, the basic component of which is the Extended Euclidean Algorithm. The Euclidean Algorithm is an effective algorithm for computing the GCD in any *Euclidean domain*.

Definition 2.2. An integral domain† R with a valuation function $v : R \setminus \{0\} \to \mathbb{N} \cup \{0\}$ is a *Euclidean Domain* if

1. for all $a, b \in R\backslash\{0\}$ we have $v(ab) > v(a)$,
2. for all $a, b \in R$ with $b \neq 0$, we can divide a by b to obtain elements $q, r \in R$ such that $a = bq + r$ where either $r = 0$ or $v(r) < v(b)$.

Polynomials q and r are called the *quotient* and the *remainder*, respectively, and the valuation function v is a *Euclidean norm function* on R. For example, if F is a field then $F[x]$, the ring of univariate polynomials over F, is a Euclidean domain with $v(a) = \deg a$.

Definition 2.3. Let R be a ring and $a, b, g \in R$. g is a *greatest common divisor* or a GCD of a and b if

(i) $g|a$ and $g|b$.

(ii) if $c|a$ and $c|b$ then $c|g$, for all $c \in R$.

In general the GCD of a and b is not unique, but all their GCDs are associates‡. Algorithm 2.3 describes how the classical Euclidean algorithm computes the GCD of two elements in a Euclidean domain. It can easily be proved that the output of this algorithm is a GCD of the inputs. Thus, the GCD is simply the last nonzero element of the remainder sequence generated by Algorithm 2.3.

---

†An Integral Domain is a commutative ring which satisfies the Cancelation Law.

‡The elements a and b are associate if $a = ub$ for $u \in R$ and u has a multiplicative inverse in R.

ALGORITHM 2.3: Classical Euclidean Algorithm (EA)

Input: $a, b \in R$, where R is a Euclidean domain.
Output: Greatest common divisor of a and b.

1. $r_0 = a, \quad r_1 = b$

2. $i = 1$
   while $r_i \neq 0$ do
   $q_i = r_{i-1}$ quo r_i /* q_i is the quotient of dividing r_{i-1} by r_i. */
   $r_{i+1} = r_{i-1} - r_i q_i$
   $i = i + 1$

3. return r_{i-1}.

The classical Euclidean algorithm can be readily extended so that it computes not only $g = \gcd(a, b)$, but also the elements s and t satisfying $sa + tb = g$. Algorithm 2.4, also called the monic Extended Euclidean Algorithm, presents the Extended Euclidean Algorithm for the Euclidean domain $F[x]$, with F a field. This algorithm makes all remainders in the remainder sequence monic, that is, to have 1 as the leading coefficient. This results in outputting a monic form of the GCD which is unique.

ALGORITHM 2.4: Extended Euclidean Algorithm (EEA)

Input: $f, g \in F[x]$, where F is a field and $\deg f \geq \deg g$.
Output: $l \in \mathbb{N}, r_i, s_i, t_i \in F[x]$, $\rho_i \in F$, for $0 \leq i \leq l+1$, and $q_i \in F[x]$ for $1 \leq i \leq l$.

1. $\rho_0 = \mathrm{lc}(f), \quad r_0 = f/\rho_0, \quad s_0 = 1/\rho_0, \quad t_0 = 0$
   $\rho_1 = \mathrm{lc}(g), \quad r_1 = g/\rho_1, \quad s_1 = 0, \quad t_1 = 1/\rho_1$

2. $i = 1$
   while $r_i \neq 0$ do

$q_i = r_{i-1} \text{ quo } r_i$
$\rho_{i+1} = \text{lc}(r_{i-1} - q_i r_i)$ /* for consistency we let $\text{lc}(0) = 1$. */
$r_{i+1} = (r_{i-1} - q_i r_i)/\rho_{i+1}$
$s_{i+1} = (s_{i-1} - q_i s_i)/\rho_{i+1}$
$t_{i+1} = (t_{i-1} - q_i t_i)/\rho_{i+1}$
$i = i + 1$

3. $l = i - 1$
return l, r_i, s_i, t_i, ρ_i for $0 \le i \le l+1$, and q_i for $1 \le i \le l$.

The elements r_i, s_i and t_i, with $0 \le i \le l+1$, are called the ith row of the Extended Euclidean Algorithm. For a better understanding of the algorithm consider the matrices

$$R_0 = \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix}, \quad Q_i = \begin{pmatrix} 0 & 1 \\ 1/\rho_{i+1} & -q_i/\rho_{i+1} \end{pmatrix}$$

in $F[x]^{2\times 2}$ and $R_i = Q_i \ldots Q_1 R_0$ for $1 \le i \le l$. From the algorithm we have

$$Q_i \begin{pmatrix} s_{i-1} & t_{i-1} \\ s_i & t_i \end{pmatrix} = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix},$$
$$Q_i \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix},$$

for $1 \le i \le l$. The following lemma presents some known properties of the Extended Euclidean Algorithm which are in the scope of this thesis.

Lemma 2.4. Let $n_i = \deg r_i$ in the EEA for inputs f and g. We let $r_0 = f/\operatorname{lc}(f), r_1 = g/\operatorname{lc}(g)$ and $r_{l+1} = 0$. Then for $0 \le i \le l$ we have

(i) $n_i > n_{i+1}$ where $i \neq 0$,

(ii) $\gcd(f, g) = \gcd(r_i, r_{i+1}) = r_l$,

(iii) $R_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}$,

(iv) $R_i \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}$,

(v) $s_i t_{i+1} - s_{i+1} t_i = \dfrac{(-1)^i}{\rho_0 \cdots \rho_{i+1}}$,

(vi) $s_i f + t_i g = r_i$; in particular, $s_l f + t_l g = \gcd(f, g)$,

(vii) $\gcd(s_i, t_i) = 1$,

(viii) $\gcd(r_i, t_i) = \gcd(f, t_i)$,

(ix) $\deg s_{i+1} = n_1 - n_i$, $\deg t_{i+1} = n_0 - n_i$ where $i \neq 0$.

Proof. (i) and (ii) are easy to show. (iii), (iv) and (v) are easily proved by induction on i. (vi) follows from (iii) and (iv). (vii) follows directly from (v).

To prove (viii), we let $g_1 = \gcd(f, t_i)$. Thus $g_1 | (s_i f + t_i g = r_i)$ or $g_1 | \gcd(r_i, t_i)$. On the other hand, if we let $g_2 = \gcd(r_i, t_i)$, then $g_2 | r_i - t_i g = s_i f$. But according to (vii) we have $\gcd(s_i, t_i) = 1$ and thus $g_2 | f$ or $g_2 | \gcd(f, t_i)$. This proves (viii).

By induction we can easily show that $\deg s_i > \deg s_{i-1}$ for $i > 1$, which implies that

$$\deg s_{i+1} = \deg(s_{i-1} - q_i s_i) = \deg q_i + \deg s_i = \sum_{j=2}^{i} \deg q_j = \sum_{j=2}^{i} (n_{j-1} - n_j) = n_1 - n_i.$$

Proof is the same for $\deg t_i$. □

Cost Analysis of the EEA

Let $f, g \in F[x]$, with F a field and $\deg f = n > \deg g = m \geq 0$. We let $n_i = \deg r_i$ for $0 \leq i \leq l + 1$, with $r_{l+1} = 0$, where r_i's are monic remainders generated by Algorithm 2.4 for inputs f and g.

To compute the quotient and the remainder of a *monic* polynomial of degree n_{i-1} divided by another *monic* polynomial of degree $n_i < n_{i-1}$, we use at most $n_i(n_{i-1} - n_i)$ multiplications and $n_i(n_{i-1} - n_i + 1)$ subtractions in F. Then to obtain a monic remainder we take one inversion plus n_{i+1} multiplications in F. So the cost of computing all quotients and monic remainders in the EEA is

$$\sum_{i=1}^{l} (2n_i(n_{i-1} - n_i) + n_i) + \sum_{i=1}^{l-1} n_{i+1} \tag{2.7}$$

subtractions and multiplications plus $l - 1$ inversions in F. It is obvious that the number of division steps l is bounded by $m + 1$. To evaluate (2.7), we consider the worst case where

the degree drops by 1 at each step, so that $n_i = m - i + 1$ for $1 \le i \le l = m + 1$. This simplifies (2.7) to

$$2m(n-m) + m + 4\sum_{i=2}^{l} n_i = 2mn - m.$$

It remains to analyze the cost for computing s_{i+1} and t_{i+1}. We can multiply the *monic* polynomial q_i by t_i using only $2\deg t_i \cdot \deg q_i + \deg q_i$ operations in F. Subtracting the product from t_{i-1} and multiplying the result by ρ_{i+1}^{-1} takes another $2(\deg t_{i+1} + 1)$ additions and multiplications. Thus the total number of additions and multiplications for computing all t_{i+1}'s $(1 \le i \le l)$ is

$$\begin{aligned}&\sum_{i=1}^{l}(\deg q_i(2\deg t_i + 1) + 2(\deg t_{i+1} + 1))\\ =&\sum_{i=1}^{l}((n_{i-1} - n_i)(2(n - n_{i-1}) + 1) + 2(n - n_i + 1))\end{aligned}$$

which simplifies to

$$3(n-m) + 2 + \sum_{i=2}^{l}(4(n-m) + 4i - 3)) = 4nm - 2m^2 + 3n + 2.$$

Using a similar argument as the one used for t_i the cost of computing all s_i's is obtained to be $2m^2 + 2m + n + 2$. Normalizing f and g in step 1 of Algorithm 2.4 also takes two inversions and $n + m$ multiplications. Thus the total cost of the EEA is at most $m + 2$ inversions and $6mn + O(n)$ additions and multiplications in F.

2.2.2 The Fast Extended Euclidean Algorithm

For the first time in 1971 Schönhage in [15] presented a fast integer GCD algorithm with time complexity $O(n \log^2 n \log\log n)$. Assuming a multiplication algorithm of time complexity $O(n \log^a n)$ is available for polynomials in $F[x]$, Moenck in [16] adapted Shoñhage's algorithm into an $O(n \log^{a+1} n)$ algorithm for polynomial GCD computation in $F[x]$. However, its correctness was restricted to input polynomials of the form "normal remainder sequences". Montgomery in his PhD thesis [17] presented a fast extended Euclidean algorithm for polynomials in $\mathbb{Z}_p[x]$ which is of $O(\mathsf{M}(n)\log n)$. Maple, Mathematica and Magma have fast integer multiplication and division. Only Magma has fast integer GCD.

As a part of this thesis, we have implemented the Fast Extended Euclidean Algorithm presented in [12, Ch. 11] for polynomials in $F[x] = \mathbb{Z}_p[x]$, with p a prime. However, the

algorithm presented in the book needs some corrections. We have made some modifications to this algorithm by removing some unnecessary outputs (Algorithm 2.5) and adding some parts for computing the quotient with maximal degree (Algorithm 3.2).

Let $f = f_n x^n + f_{n-1} x^{n-1} + \ldots + f_0 \in F[x]$ and $f_n \neq 0$. Following [12], we define the truncated polynomial

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

for $k \in \mathbb{Z}$. We set $f_i = 0$ if $i < 0$ and $f \upharpoonright k = 0$ if $k < 0$. The polynomial $f \upharpoonright k$ is of degree k for $k \geq 0$ and its coefficients are the $k + 1$ highest coefficients of f.

Definition 2.5. The pairs (f, g) and (f^*, g^*) *coincide up to* k if

$$\begin{aligned} f \upharpoonright k &= f^* \upharpoonright k, \\ g \upharpoonright (k - (\deg f - \deg g)) &= g^* \upharpoonright (k - (\deg f^* - \deg g^*), \end{aligned} \tag{2.8}$$

where $f, g, f^*, g^* \in F[x] \backslash \{0\}$, $\deg f > \deg g$, $\deg f^* > \deg g^*$ and $k \in \mathbb{Z}$. If $k \geq \deg f - \deg g$, then $\deg f - \deg g = \deg f^* - \deg g^*$.

Example 2.6. Consider $f = 2x^8 + x^7 + 4x^5 + 3x^2 + 1$, $g = x^7 + 5x^5 + 3x^4 + x^2 + 6$ and $f^* = 2x^7 + x^6 + 4x^4 + 3x + 5$, $g^* = x^6 + 5x^4 + 3x^3 + x$. Then (f, g) and (f^*, g^*) coincide up to 6 because

$$\begin{aligned} f \upharpoonright 6 = f^* \upharpoonright 6 &= 2x^6 + x^5 + 4x^3 + 3 \\ g \upharpoonright 5 = g^* \upharpoonright 5 &= x^5 + 5x^3 + 3x^2 + 1 \end{aligned}$$

Lemma 2.7. [12, Lemma 11.1] Let $k \in \mathbb{Z}$, $f, g, f^*, g^* \in F[x] \backslash \{0\}$. If (f, g) and (f^*, g^*) coincide up to $2k$ and $k \geq \deg f - \deg g$ then

1. $q = q^*$ and
2. if $r \neq 0$ and $k - \deg q \geq \deg g - \deg r$ then (g, r) and (g^*, r^*) coincide up to $2(k - \deg q)$, where $q, r, q^*, r^* \in F[x]$ are defined by

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

Lemma 2.7 gives the requirements necessary for the quotients to be equal. Refer to [12, Lemma 11.1] for the proof.

Let $n_i = \deg r_i$ for $0 \leq i \leq l+1$ and $r_{l+1} = 0$, where r_i's are monic polynomials in the remainder sequence generated by the Euclidean Algorithm for monic polynomials r_0 and r_1. We let $m_i = \deg q_i = n_{i-1} - n_i$ for $1 \leq i \leq l$, where q_i is the ith quotient in the Euclidean Algorithm. Then we have

$$n_0 - n_j = \sum_{i=1}^{j} m_i. \tag{2.9}$$

For any $k \in \mathbb{N}$ and $f, g \in F[x]$, define the positive integer number $\eta_{f,g}(k)$ by

$$\eta_{f,g}(k) = \max_{0 \leq j \leq l} \{j : \sum_{i=1}^{j} m_i \leq k\}. \tag{2.10}$$

The following inequality is derived from (2.9) and (2.10),

$$\sum_{i=1}^{\eta_{f,g}(k)} m_i = n_0 - n_{\eta_{f,g}(k)} \leq k < n_0 - n_{\eta_{f,g}(k)+1} = \sum_{i=1}^{\eta_{f,g}(k)+1} m_i. \tag{2.11}$$

Lemma 2.8. [12, Lemma 11.3] Let $k \in \mathbb{N}$, $h = \eta_{r_0,r_1}(k)$ and $h^* = \eta_{r_0^*,r_1^*}(k)$, with r_0, r_1, r_0^*, r_1^* monic polynomials in $F[x]$. If (r_0, r_1) and (r_0^*, r_1^*) coincide up to $2k$ and $k \geq \deg r_0 - \deg r_1$, then

1. $h = h^*$,
2. $q_i = q_i^*$ for $1 \leq i \leq h$,
3. $\rho_i = \rho_i^*$ for $2 \leq i \leq h$,

where $q_i, q_i^* \in F[x]$ and $\rho_i, \rho_i^* \in F$ are defined by

$$\begin{aligned} r_{i-1} &= q_i r_i + \rho_{i+1} r_{i+1} \qquad (1 \leq i \leq l), \quad r_{l+1} = 0, \\ r_{i-1}^* &= q_i^* r_i^* + \rho_{i+1}^* r_{i+1}^* \qquad (1 \leq i \leq l^*), \quad r_{l^*+1}^* = 0. \end{aligned}$$

Remark 2.9. Note that the original lemma in [12] states that $\rho_{h+1} = \rho_{h+1}^*$ which is not correct and we have excluded $h+1$ in Lemma 2.8.

The proof for Lemma 2.8 follows directly from Lemma 2.7. Refer to [12, Lemma 11.3] for a detailed proof of this lemma. To improve the efficiency of the EEA, a divide-and-conquer

algorithm is designed based on Lemma 2.8. This algorithm is called the *Fast Extended Euclidean Algorithm* and is presented as Algorithm 2.5.

Algorithm 2.5 works by dividing the sequence of the quotients into two parts such that the sum of the degrees in both parts is almost the same. Let l be the number of division steps in the Euclidean algorithm. Then in the case of a normal degree sequence, in which the quotient degree drops exactly by 1 at each step, the problem is divided into two subproblems of size $l/2$.

To obtain the quotient of the division of a large monic polynomial r_0 by another large monic polynomial r_1, one can divide two smaller polynomials r_0^* and r_1^*, provided that (r_0, r_1) and (r_0^*, r_1^*) coincide up to $2k$, where $k \geq \deg r_0 - \deg r_1$. This can even be extended to applying the Euclidean Algorithm on r_0^*, r_1^* instead of r_0, r_1 and get the same first $\eta_{r_0,r_1}(k)$ quotients, and the same first $\eta_{r_0,r_1}(k) - 1$ leading coefficients of the remainders, by Lemma 2.8.

Algorithm 2.5 gets two *monic* polynomials r_0, r_1 and a positive integer k as input, with $n_0/2 \leq k \leq n_0$. Input k helps us divide the problem into two subproblems of almost the same size $(k/2)$. The FEEA is then recursively applied to solve each problem. The sum of degrees of the quotients computed in each call to the FEEA is less than or equal to k. That is, if we let $m_i = \deg q_i$ then it should return whenever $\sum_{i=1}^{h+1} m_i > k$. According to (2.10) $h = \eta_{r_0,r_1}(k)$. If the algorithm is called with a value of k which satisfies the condition $n_0/2 \leq k \leq n_0$, then in any further call to the FEEA we will have $k = n_0/2$. We will explore the special cases where $0 < k < n_0/2$, or inputs r_0 and r_1 are not monic or $\deg r_0 = \deg r_1$ later.

The following four items describe the four outputs of Algorithm 2.5:

- $h = \eta_{r_0,r_1}(k)$ specifies the total number of steps of the Extended Euclidean Algorithm performed in one call to Algorithm 2.5. Note that the FEEA computes all the elements of the EEA except the remainders.

- ρ_{h+1} is the leading coefficient of the $(h+1)$th remainder in the Extended Euclidean Algorithm, that is $\rho_{h+1} r_{h+1} = r_{h-1} - r_h q_h$ where r_{h-1}, r_h and r_{h+1} are all monic polynomials.

- $R_h = \begin{pmatrix} s_h & t_h \\ s_{h+1} & t_{h+1} \end{pmatrix}$ is a matrix that helps us compute the monic remainders r_h and r_{h+1} from r_0 and r_1, in addition to holding the values of $s_h, t_h, s_{h+1}, t_{h+1}$.

ALGORITHM 2.5: Fast Extended Euclidean Algorithm (FEEA)

Input: r_0, r_1 two *monic* polynomials in $F[x]$ with $n_0 = \deg r_0 > n_1 = \deg r_1 \geq 0$ and $k \in \mathbb{N}$ with $n_0/2 \leq k \leq n_0$. /* n_0 is strictly greater than n_1. */

Output: $h = \eta_{r_0,r_1}(k) \in \mathbb{N}$, $\rho_{h+1} \in F$, $R_h = \begin{pmatrix} s_h & t_h \\ s_{h+1} & t_{h+1} \end{pmatrix}$ and $\begin{pmatrix} r_h \\ r_{h+1} \end{pmatrix} = R_h \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$.

1. if $r_1 = 0$ or $k < n_0 - n_1$ then
   return $0, 1, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ and $\begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$
   else if $n_0 <$ *cutoff* then /* *cutoff* degree of the FEEA */
   return EEA(r_0, r_1, k)

2. $k_1 = \lfloor k/2 \rfloor$

3. $r_0^* = r_0 \upharpoonright 2k_1, \quad r_1^* = r_1 \upharpoonright (2k_1 - (n_0 - n_1))$
   call the algorithm recursively by writing FEEA(r_0^*, r_1^*,k_1), to obtain
   $j - 1 = \eta_{r_0^*,r_1^*}(k_1)$, ρ_j^*, $R_{j-1}^* = Q_{j-1}^* Q_{j-2} \ldots Q_1$ where $Q_{j-1}^* = \begin{pmatrix} 0 & 1 \\ \frac{1}{\rho_j^*} & \frac{-q_{j-1}}{\rho_j^*} \end{pmatrix}$,
   and $\begin{pmatrix} r_{j-1}^* \\ r_j^* \end{pmatrix} = R_{j-1}^* \begin{pmatrix} r_0^* \\ r_1^* \end{pmatrix}$

4. /* in this step we want to determine ρ_j, r_{j-1}, r_j and R_{j-1}. */
   $\begin{pmatrix} r_{j-1} \\ \tilde{r}_j \end{pmatrix} = R_{j-1}^* \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$
   $R_{j-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1/\operatorname{lc}(\tilde{r}_j) \end{pmatrix} R_{j-1}^*$
   $\rho_j = \rho_j^* \operatorname{lc}(\tilde{r}_j), \quad r_j = \tilde{r}_j / \operatorname{lc}(\tilde{r}_j)$

5. $\begin{pmatrix} n_{j-1} \\ n_j \end{pmatrix} = \begin{pmatrix} \deg r_{j-1} \\ \deg r_j \end{pmatrix}$
   if $r_j = 0$ or $k < n_0 - n_j$ then
   return $j - 1$, ρ_j, R_{j-1}, $\begin{pmatrix} r_{j-1} \\ r_j \end{pmatrix}$

6. $q_j = r_{j-1} \operatorname{quo} r_j$
   $\rho_{j+1} = \operatorname{lc}(r_{j-1} - q_j r_j)$ /* for consistency we let $\operatorname{lc}(0) = 1$. */
   $r_{j+1} = (r_{j-1} - q_j r_j)/\rho_{j+1}$

$n_{j+1} = \deg r_{j+1}$

$R_j = \begin{pmatrix} 0 & 1 \\ 1/\rho_{j+1} & -q_j/\rho_{j+1} \end{pmatrix} R_{j-1}$

7. $k_2 = k - (n_0 - n_j)$ /* up to now we have computed j quotients. */

8. $r_j^* = r_j \upharpoonright 2k_2, \quad r_{j+1}^* = r_{j+1} \upharpoonright (2k_2 - (n_j - n_{j+1}))$
   call the algorithm recursively by writing FEEA(r_j^*, r_{j+1}^*, k_2), to obtain
   $h - j = \eta_{r_j^*, r_{j+1}^*}(k_2)$, ρ_{h+1}^*, $\tilde{S} = Q_h^* Q_{h-1} \ldots Q_{j+1}$ where $Q_h^* = \begin{pmatrix} 0 & 1 \\ \dfrac{1}{\rho_{h+1}^*} & \dfrac{-q_h}{\rho_{h+1}^*} \end{pmatrix}$,
   and $\begin{pmatrix} r_h^* \\ r_{h+1}^* \end{pmatrix} = \tilde{S} \begin{pmatrix} r_j^* \\ r_{j+1}^* \end{pmatrix}$

9. $\begin{pmatrix} r_h \\ \tilde{r}_{h+1} \end{pmatrix} = \tilde{S} \begin{pmatrix} r_j \\ r_{j+1} \end{pmatrix}$
   $S = \begin{pmatrix} 1 & 0 \\ 0 & 1/\operatorname{lc}(\tilde{r}_{h+1}) \end{pmatrix} \tilde{S}$
   $\rho_{h+1} = \rho_{h+1}^* \operatorname{lc}(\tilde{r}_{h+1}), \quad r_{h+1} = \tilde{r}_{h+1}/\operatorname{lc}(\tilde{r}_{h+1})$

10. return h, ρ_{h+1}, SR_j, $\begin{pmatrix} r_h \\ r_{h+1} \end{pmatrix}$

- $\begin{pmatrix} r_h \\ r_{h+1} \end{pmatrix} = R_h \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$ is a vector containing the hth and the $(h+1)$th monic remainders in the Extended Euclidean Algorithm. If h is equal to the total number of steps of the Extended Euclidean Algorithm on r_0 and r_1 i.e. l, then $r_h = \gcd(r_0, r_1)$ and $r_{h+1} = 0$.

In step 3 of Algorithm 2.5, a recursive call is made with $k_1 = \lfloor k/2 \rfloor$ as the integer input, so that when completed $\eta_{r_0^*, r_1^*}(k_1) = j-1$ quotients have been computed. This is almost half of the quotients in the case of a normal degree sequence. The pairs (r_0, r_1) and (r_0^*, r_1^*) coincide up to $2k_1$ and $n_0 - n_1 \leq k$, thus according to Lemma 2.8 $\eta_{r_0, r_1}(k_1) = \eta_{r_0^*, r_1^*}(k_1) = j - 1$, $q_i = q_i^*$ for $1 \leq i \leq j-1$ and $\rho_i = \rho_i^*$ for $2 \leq i \leq j-1$. Note that ρ_j is not necessarily equal to ρ_j^*. In step 6 we compute the next quotient q_j, and then in step 8 another recursive call is made with $k_2 = k - (n_0 - n_j) = k - \sum_{i=1}^{j} \deg q_i$. This will perform the rest of the divisions and when completed all the expected quotients have been computed.

In step 4 we obtain the values of ρ_j, r_{j-1}, r_j and R_{j-1}. We have

$$R^*_{j-1} = Q^*_{j-1} R_{j-2} = \begin{pmatrix} 0 & 1 \\ \dfrac{1}{\rho^*_j} & \dfrac{-q_{j-1}}{\rho^*_j} \end{pmatrix} \begin{pmatrix} s_{j-2} & t_{j-2} \\ s_{j-1} & t_{j-1} \end{pmatrix} = \begin{pmatrix} s_{j-1} & t_{j-1} \\ \dfrac{\rho_j}{\rho^*_j} s_j & \dfrac{\rho_j}{\rho^*_j} t_j \end{pmatrix}, \tag{2.12}$$

hence

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

Let $\tilde{r}_j = \dfrac{\rho_j}{\rho^*_j} r_j$. Since r_j is monic we get

$$\begin{aligned} \rho_j = \rho^*_j \operatorname{lc}(\tilde{r}_j) \quad &\Longrightarrow \quad r_j = \tilde{r}_j / \operatorname{lc}(\tilde{r}_j), \\ R_{j-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1/\operatorname{lc}(\tilde{r}_j) \end{pmatrix} & R^*_{j-1} = Q_{j-1} \dots Q_1. \end{aligned}$$

As stated before $\eta_{r_0,r_1}(k_1) = j-1$, thus according to (2.11) we have

$$n_0 - n_{j-1} \le k_1 < n_0 - n_j.$$

If now $k < n_0 - n_j$ in step 5, then $\eta_{r_0,r_1}(k) = j-1$ and the algorithm returns the correct result; otherwise, $\eta_{r_0,r_1}(k) \ge j$ and the execution is continued with computing the next quotient q_j in step 6.

In step 8 after the recursive call we have $\tilde{S} = Q^*_h Q_{h-1} \dots Q_{j+1}$, and by Lemma 2.4 (iv) we obtain

$$\begin{aligned} \tilde{S} \begin{pmatrix} r_j \\ r_{j+1} \end{pmatrix} &= \tilde{S}.R_j \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = Q^*_h R_{h-1} \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = Q^*_h \begin{pmatrix} r_{h-1} \\ r_h \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ \dfrac{1}{\rho^*_{h+1}} & \dfrac{-q_h}{\rho^*_{h+1}} \end{pmatrix} \begin{pmatrix} r_{h-1} \\ r_h \end{pmatrix} = \begin{pmatrix} r_h \\ \dfrac{\rho_{h+1}}{\rho^*_{h+1}} r_{h+1} \end{pmatrix}. \end{aligned}$$

Let $\tilde{r}_{h+1} = \dfrac{\rho_{h+1}}{\rho^*_{h+1}} r_{h+1}$. By analogous reasoning as for step 4 we get

$$\begin{aligned} \rho_{h+1} = \rho^*_{h+1} \operatorname{lc}(\tilde{r}_{h+1}) \quad &\Longrightarrow \quad r_{h+1} = \tilde{r}_{h+1} / \operatorname{lc}(\tilde{r}_{h+1}), \\ S = \begin{pmatrix} 1 & 0 \\ 0 & 1/\operatorname{lc}(\tilde{r}_{h+1}) \end{pmatrix} & \tilde{S} = Q_h \dots Q_{j+1}. \end{aligned}$$

As stated before, the inputs of the FEEA are expected to be two monic polynomials r_0, r_1 and a positive integer k, where $\deg r_0 > \deg r_1$ and $n_0/2 \leq k \leq n_0$. Let $f, g \in F[x]$ be two arbitrary polynomials and $k \in \mathbb{N}$. We now explain how to handle the special cases that might occur.

1. f and g are monic, but $\deg f = \deg g$:
   Let $\rho_2 = \mathrm{lc}(f - g)$. If $f = g$ then we let $\rho_2 = \mathrm{lc}(0) = 1$. We can call the FEEA with $r_0 = g, r_1 = (f - g)/\rho_2$ and then instead of $R_h = \begin{pmatrix} s_h & t_h \\ s_{h+1} & t_{h+1} \end{pmatrix}$ return the following matrix
   $$R_h \begin{pmatrix} 0 & 1 \\ 1/\rho_2 & -1/\rho_2 \end{pmatrix} = \begin{pmatrix} t_h/\rho_2 & s_h - t_h/\rho_2 \\ t_{h+1}/\rho_2 & s_{h+1} - t_{h+1}/\rho_2 \end{pmatrix}.$$
   The subtraction, normalization and the corresponding corrections of R_h cost only $O(n)$ field operations and hence do not affect the asymptotic running time of the algorithm.

2. $\deg f > \deg g$, but f and g are not monic:
   We run the algorithm on $r_0 = f/\mathrm{lc}(f), r_1 = g/\mathrm{lc}(g)$ and divide the first and the second column of the result R_h by $\mathrm{lc}(f)$ and $\mathrm{lc}(g)$, respectively. This takes only $O(n)$ additional field operations.

3. $0 < k < n_0/2$:
   It suffices to call the algorithm with input $r_0 \upharpoonright 2k$, $r_1 \upharpoonright (2k - (\deg r_0 - \deg r_1))$ and k, and make the same corrections on ρ_{h+1}, r_{h+1} and R_h as we did in step 4 of the algorithm.

Now one question is, what value should we choose for input k, when we want to compute $\gcd(r_0, r_1)$ using the FEEA? The output $h = \eta_{r_0,r_1}(k)$ denotes the number of steps of the EEA performed, or equivalently, the number of quotients computed in the FEEA with inputs r_0, r_1 and k. Let l denote the total number of steps of the EEA. We have

$$\sum_{i=1}^{l} \deg q_i \leq \deg r_0.$$

If we set $k = \deg r_0$, then $h = \eta_{r_0,r_1}(\deg r_0) = l$ which results in computing all the quotients or $\gcd(r_0, r_1)$ as well.

Cost Analysis of the FEEA

Let $T(k)$ denote the number of additions and multiplications that Algorithm 2.5 performs in F with input k. Step 3 of the algorithm performs $T(k_1) = T(\lfloor k/2 \rfloor)$ operations for solving a subproblem of the same kind. Definition of k_2 and inequality (2.11) imply that

$$k_2 = k - (n_0 - n_j) < k - k_1 = \lceil k/2 \rceil$$

or $k_2 \leq \lfloor k/2 \rfloor$. Thus step 8 takes $T(k_2)$ or at most $T(\lfloor k/2 \rfloor)$ operations in F.

To make the algorithm work more efficiently, in step 4 instead of multiplying R^*_{j-1} by $(r_0, r_1)^T$ whose entries are at most of degree $2k$, we multiply it by a vector with entries of degree at most $n_0 - 2k_1 - 1 \leq k$ as follows

$$\begin{aligned} R^*_{j-1}\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} &= R^*_{j-1}\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} - \left(R^*_{j-1}\begin{pmatrix} r^*_0 \\ r^*_1 \end{pmatrix} - \begin{pmatrix} r^*_{j-1} \\ r^*_j \end{pmatrix} \right) x^{n_0 - 2k_1} \\ &= R^*_{j-1}\begin{pmatrix} r_0 - r^*_0 x^{n_0-2k_1} \\ r_1 - r^*_1 x^{n_0-2k_1} \end{pmatrix} + \begin{pmatrix} r^*_{j-1} x^{n_0-2k_1} \\ r^*_j x^{n_0-2k_1} \end{pmatrix}. \end{aligned}$$

The entries of $R^*_{j-1} = \begin{pmatrix} s_{j-1} & t_{j-1} \\ \rho_j/\rho^*_j s_j & \rho_j/\rho^*_j t_j \end{pmatrix}$ are of degrees $n_1 - n_{j-2}$, $n_0 - n_{j-2}$, $n_1 - n_{j-1}$ and $n_0 - n_{j-1}$, by (2.12) and Lemma 2.4 (ix). All four values are at most $n_0 - n_{j-1} \leq k_1 = \lfloor k/2 \rfloor$. Thus we have four multiplications of polynomials of degree at most $\lfloor k/2 \rfloor$ by polynomials of degree less than or equal to k, plus some multiplications by constants and some additions. Thus the cost for step 4 is $4\mathsf{M}(k) + O(k)$.

In step 9 instead of multiplying $\tilde{S}$ by $(r_j, r_{j+1})^T$ we do the same computations as we did in step 4 to get the result more efficiently. The entries of $\tilde{S}$ are of degrees $n_{j+1} - n_{h-1}$, $n_j - n_{h-1}$, $n_{j+1} - n_h$ and $n_j - n_h$ which are at most $n_j - n_h \leq k_2 \leq \lfloor k/2 \rfloor$. The polynomials in the vector to which $\tilde{S}$ is applied are of degree $n_j - 2k_2 - 1 = n_0 - k - k_2 - 1 < k$. Thus as step 4, the cost for step 9 is bounded by $4\mathsf{M}(k) + O(k)$.

In step 6 we divide r_{j-1} by r_j and compute the quotient q_j and the remainder r_{j+1} of this division. Polynomial r_j is of degree $n_j < n_0 \leq 2k$ and the quotient q_j is of degree $n_{j-1} - n_j \leq n_0 - (n_0 - k) = k$. Fast division as explained in [12, Algorithm 9.5] takes $4\mathsf{M}(k) + O(k)$ operations in F for computing the quotient and at most $2\mathsf{M}(k) + O(k)$ operations in F for computing the remainder on inputs r_{j-1} and r_j. So the cost of performing the division would be $6\mathsf{M}(k) + O(k)$.

Remark 2.10. We did not implement Fast Division in our implementation of the FEEA. The Fast Division Algorithm is needed when the degree of the quotient is not small, but in the normal case where the degree drops by a small amount at each step of the FEEA there is no need to use this algorithm.

Another computation performed in step 6 is to compute R_j, the first row of which is exactly the same as the second row of R_{j-1}. Thus, we only want to compute $s_{j+1} = (s_{j-1} - s_j q_j)/\rho_{j+1}$ and $t_{j+1} = (t_{j-1} - t_j q_j)/\rho_{j+1}$ as the elements of the second row of R_j. As stated before s_j and t_j are at most of degree $\lfloor k/2 \rfloor$ and q_j is at most of degree k, which implies computing the elements of the second row of R_j takes at most $2\mathsf{M}(k) + O(k)$ operations in F.

The entries in the first row of $R_j = \begin{pmatrix} s_j & t_j \\ s_{j+1} & t_{j+1} \end{pmatrix}$ are of degree $\lfloor k/2 \rfloor$ and the entries in the second row are at most of degree $n_0 - n_j \leq k$. Also as shown before, the degrees of the entries of S are at most $\lfloor k/2 \rfloor$. Thus computation of $S.R_j$ in step 10, takes at most $6\mathsf{M}(k) + O(k)$ operations in F.

The only inversions that take place in Algorithm 2.5 are $1/\operatorname{lc}(\tilde{r}_j)$, $1/\rho_{j+1}$ and $1/\operatorname{lc}(\tilde{r}_{h+1})$. They all can be computed only once. Therefore the total number of inversions during the recursive process is at most $3k$. The following table illustrates the cost of each step of the FEEA.

Step	Cost
3	$T(\lfloor k/2 \rfloor)$
4	$4\mathsf{M}(k) + O(k)$
6	$8\mathsf{M}(k) + O(k)$
8	$T(\lfloor k/2 \rfloor)$
9	$4\mathsf{M}(k) + O(k)$
10	$6\mathsf{M}(k) + O(k)$
Total	$2T(\lfloor k/2 \rfloor) + 22\mathsf{M}(k) + O(k)$

Table 2.2: The number of multiplications and additions of steps of Algorithm 2.5

T satisfies the following recursive inequalities

$$T(0) = 0, \quad T(k) \leq 2T(\lfloor k/2 \rfloor) + 22\mathsf{M}(k) + ck,$$

for some constant $c \in \mathbb{R}$.

Hence

$$T(k) \leq (22\mathsf{M}(k) + O(k)) \log k \in O(\mathsf{M}(k) \log k).$$

We used Karatsuba's multiplication algorithm in our implementation of the Fast Extended Euclidean Algorithm. In this case $\mathsf{M}(k) \in O(k^{\log_2 3})$ and thus the implemented FEEA is of time complexity $O(k^{\log_2 3} \log k)$. The EEA performs better than the FEEA for polynomials of low degrees. Thus we have computed a *cutoff* degree for the dividend r_0 below which we use the EEA in Algorithm 2.5. Our implementation of the EEA (in Java) accepts 3 inputs and returns the same number of outputs as the FEEA, so that it can be used in step 1 of the FEEA.

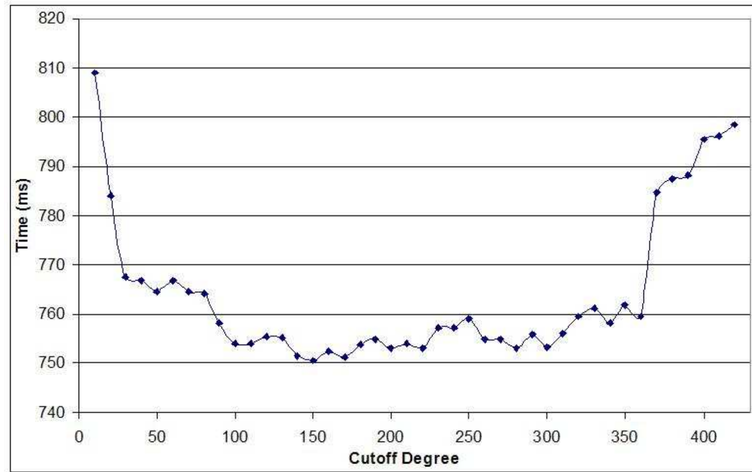

Figure 2.4: Timings (in ms) of the FEEA for different cutoff degree

Figure 2.4 illustrates the timings (in ms) of the FEEA on two random polynomials of degree 10000. We can choose 150 as the cutoff degree, but it seems that any value in the range 100 to 300 can be chosen as the cutoff degree.

Table 2.3 illustrates some timings for the EEA and the FEEA using cutoff degree 150. The first column (n) specifies the degree of the two randomly chosen polynomials. The second and the third columns show the time it takes to run the EEA and the FEEA, respectively, on input polynomials of degree n. We have divided the timings presented in the third column by $n^{\log_2 3} \log n$ for each value of n and obtained a constant factor in the fourth column, which confirms that our implementation of the FEEA is of $O(n^{\log_2 3} \log n)$.

n	EEA(ms)	FEEA(ms)	FEEA/($n^{\log_2 3}\log n$)	EEA/FEEA
1000	373.80	295.63	0.00052	1.26
2000	1427.18	942.83	0.00050	1.51
4000	5602.18	2972.08	0.00049	1.88
8000	22295.47	9588.76	0.00048	2.33
16000	88766.90	31278.50	0.00049	2.84
32000	354085.71	99273.77	0.00048	3.54

Table 2.3: Timings (in ms) of the FEEA compared to the EEA

All our computations were performed modulo a 15 bit prime.

2.2.3 Fast Polynomial Interpolation (Application)

To complete this chapter we show how the FEEA can be applied to solve the polynomial interpolation problem fast by using a more or less obvious divide and conquer algorithm. Let F be a field and $\alpha_1, \ldots, \alpha_n \in F$ be pairwise distinct. Given arbitrary $\beta_1, \ldots, \beta_n \in F$, we want to find $f \in F[x]$ of degree less than n such that $f(\alpha_i) = \beta_i$ for $i = 1 \ldots n$. It is well known that if α_i's are distinct a solution exists and is unique. The solution can be found by solving a system of linear equations. Let $f(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0$. Then we have

$$f(\alpha_i) = \beta_i = a_{n-1}\alpha_i^{n-1} + \ldots + a_1\alpha_i + a_0, \quad i = 1 \ldots n.$$

The system can be solved in $O(n^3)$ operations in F. It is also well known that the problem can be solved in $O(n^2)$ operations in F using either *Lagrange* or *Newton* Interpolation. Here we show how to use the FEEA to generate a divide-and-conquer algorithm for fast polynomial interpolation. Description of the algorithm follows:

1. Find f_1 the polynomial interpolating $\alpha_1, \ldots, \alpha_{n/2}$ by recursively calling the algorithm. Let $m_1(x) = \prod_{i=1}^{n/2}(x - \alpha_i)$. Then $f_1(x)$ and $m_1(x)$ satisfy $f(x) \equiv f_1(x) \mod m_1(x)$.

2. Find f_2 the polynomial interpolating $\alpha_{n/2+1}, \ldots, \alpha_n$ by recursively calling the algorithm. Let $m_2(x) = \prod_{i=n/2+1}^{n}(x - \alpha_i)$. Then $f_2(x)$ and $m_2(x)$ satisfy $f(x) \equiv f_2(x) \mod m_2(x)$.

3. Find f using the Chinese Remainder Algorithm. Let

$$f(x) = v_1(x) + v_2(x)m_1(x),$$

where $0 \leq \deg v_i < \deg m_i$. Thus we have

$$f_1(x) \equiv v_1(x) \mod m_1(x),$$
$$f_2(x) \equiv v_1(x) + v_2(x)m_1(x) \mod m_2(x).$$

If we choose $v_1(x) = f_1(x)$, then we can compute

$$v_2(x) = (f_2(x) - f_1(x))/m_1(x) \mod m_2(x)$$

by solving $sm_1 + tm_2 = 1$ for s, to find the inverse of $m_1(x)$ mod $m_2(x)$, using the FEEA.

All f_1, f_2, m_1 and m_2 are at most of degree $n/2$, thus computing v_2 needs one application of the FEEA to compute $1/m_1(x) \mod m_2(x)$ which takes $O(\mathsf{M}(n/2)\log(n/2))$ field operations and one multiplication of $O(\mathsf{M}(n/2))$ field operations. Let $T(n)$ denote the cost of computing $f(x)$, the polynomial interpolating n distinct points using the fast interpolation algorithm explained above. Then we have

$$T(n) = 2T(n/2) + 2\mathsf{M}(n/2) + O(\mathsf{M}(n/2)\log(n/2)).$$

Hence

$$T(n) \in O(\mathsf{M}(n)\log^2 n).$$

Remark 2.11. Fast Polynomial Interpolation can be even done in $O(\mathsf{M}(n)\log n)$ using the algorithm described in [12, Sec. 10.2].

In the following chapter we will describe the application of polynomial interpolation in Rational Function Reconstruction.

Chapter 3

Rational Function Reconstruction

The general problem of *rational reconstruction* consists of *rational number reconstruction* and *rational function reconstruction* problems. The former problem reconstructs a rational number (in $\mathbb{Q}$) which is congruent to some integer modulo another integer, while the latter reconstructs a rational function that is congruent to some polynomial modulo another polynomial. We will address the second problem in this chapter.

In Section 3.1 we describe the rational function interpolation problem and in Section 3.2 we introduce two solutions for the rational function reconstruction problem: Wang's algorithm and a fast maximal quotient algorithm.

3.1 Rational Function Interpolation (Cauchy Interpolation)

Rational Function Interpolation, also called *Cauchy Interpolation*, is the most general form of polynomial interpolation. Let F be a field and $\alpha_1, \ldots, \alpha_m \in F$ be pairwise distinct. Given arbitrary $\beta_1, \ldots, \beta_m \in F$, we are looking for a rational function $f = n/d \in F(x)$, with $n, d \in F[x]$, such that

$$d(\alpha_i) \neq 0, \quad f(\alpha_i) = \frac{n(\alpha_i)}{d(\alpha_i)} = \beta_i \quad 1 \leq i \leq m.$$

We want the rational function f to be in canonical form, that is, d to be monic and $\gcd(n, d) = 1$. Yet, f is not unique since n and d should only satisfy $\deg n + \deg d < m$. Solving a system of equations and Wang's algorithm are the two solutions we describe in this section for the Rational Function Interpolation problem.

The System of Equations

Let α_i and β_i be defined as above and $f = n/d$ be the rational function we want to find. To have a unique solution for f we let the denominator to be of the given degree $0 \leq k < m$. Then the numerator would be at most of degree $m - k - 1$. Thus we let

$$n(x) = a_{m-k-1}x^{m-k-1} + \ldots + a_1 x + a_0,$$
$$d(x) = x^k + b_{k-1}x^{k-1} + \ldots + b_1 x + b_0.$$

The rational function $f = n/d$ is obtained by solving the following system of equations

$$n(\alpha_i) = d(\alpha_i)\beta_i \quad d(\alpha_i) \neq 0 \quad i = 1 \ldots m,$$

for the coefficients of n and d. Using Gaussian elimination it takes $O(m^3)$ operations in F.

Using Wang's Algorithm

In the second solution, we first find the unique interpolating polynomial $g \in F[x]$ of degree less than m such that $g(\alpha_i) = \beta_i$ for $i = 1 \ldots m$. Thus we will have

$$f(x) = \frac{n(x)}{d(x)} \equiv g(x) \mod (x - \alpha_i),\ d(\alpha_i) \neq 0 \quad \text{for } i = 1 \ldots m. \tag{3.1}$$

Let $M(x) = \prod_{i=1}^{m}(x - \alpha_i)$. Then (3.1) is equivalent to

$$f(x) = \frac{n(x)}{d(x)} \equiv g(x) \mod M(x),\ \gcd(M, d) = 1. \tag{3.2}$$

Now the problem is, given polynomial $M(x)$ of degree m and polynomial g of degree less than m, find the rational function $f = n/d$ satisfying (3.2). This problem is called the rational function reconstruction problem. We describe the solutions to this problem in Section 3.2. Computing g using Newton interpolation takes $O(m^2)$ operations in F. The cost of computing M is $O(\mathsf{M}(m) \log m)$. If we use Wang's algorithm to reconstruct the rational function $f = n/d$ which uses the Extended Euclidean Algorithm, then the cost of computing f would be of $O(m^2)$ operations in F. Thus the total cost would be of $O(m^2)$ operations in F.

Remark 3.1. In Section 3.2.1 we briefly describe how the FEEA can be modified so that it can be used by Wang's algorithm for recovering a rational function. Also we can use the fast polynomial interpolation algorithm described in [12] to compute g. This will result in an algorithm taking $O(\mathsf{M}(m) \log m)$ operations in F.

3.2 Rational Function Reconstruction (RFR)

Let $F = \mathbb{Z}_{11}$. We want to compute a rational function $n/d \in F(x)$ where $n/d \equiv g \mod f$, with $f = \prod_{i=1}^{7}(x-i)$ and $g = x^6 + 3x^5 + 8x^4 + 4x^3 + 6x^2 + x + 9$. If we run the Extended Euclidean Algorithm with input f and g, then according to Lemma 2.4 (vi) we have $fs_i + gt_i = r_i$ or, equivalently, $r_i \equiv gt_i \mod f$. Now if $\gcd(f, t_i) = 1$ then we have $r_i/t_i \equiv g \mod f$. This implies that (n, d) can be equal to any pair of (r_i, t_i), generated by the EEA, provided that $\gcd(f, t_i) = 1$. The following table illustrates the values of r_i, t_i and q_i in each iteration of the Extended Euclidean Algorithm for given inputs f and g defined above.

i	r_i	q_i	t_i
0	$x^7 + 5x^6 + 3x^5 + 9x^4 + 4x^3 + 2x^2 + 9$	$-$	0
1	$x^6 + 3x^5 + 8x^4 + 4x^3 + 6x^2 + x + 9$	$x + 2$	1
2	$x^3 + 2$	$x^3 + 3x^2 + 8x + 2$	$10x + 9$
3	$7x + 5$	$8x^2 + 10x + 7$	$x^4 + 5x^3 + 3x^2 + 7x + 5$

From row 1, 2 and 3 we get the following solutions

$$\frac{r_1}{t_1} = \frac{x^6 + 3x^5 + 8x^4 + 4x^3 + 6x^2 + x + 9}{1},$$
$$\frac{r_2}{t_2} = \frac{x^3 + 2}{10x + 9} = \frac{10x^3 + 9}{x + 2},$$
$$\frac{r_3}{t_3} = \frac{7x + 5}{x^4 + 5x^3 + 3x^2 + 7x + 5}.$$

We seek a way to choose one rational function among all possible solutions. It is not hard to see that if we want to recover a rational function with $\deg n \le N$ and $\deg d \le D$, then we must have $N + D < \deg f$.

Let $M = \deg f$. Wang in [1] gave a solution to the rational number reconstruction problem. His algorithm can be readily extended for the rational functions as well, by setting $N = \lfloor M/2 \rfloor$ and $D = M - N - 1$. We will describe Wang's algorithm for rational functions in Section 3.2.1. Thus if we use Wang's algorithm the solution to the above example would be $r_2/t_2 = (10x^3 + 9)/(x + 2)$. In Section 3.2.2 we introduce a fast algorithm for solving the rational function reconstruction problem. This algorithm outputs the rational function with the smallest total degree ($\deg n + \deg d$) provided that $\deg n + \deg d < \deg f - 1$. Thus the output of this algorithm for the example presented above is the same as Wang's output. The following lemma gives us some hint on the general solutions to the RFR problem.

Lemma 3.2. (Uniqueness of the EEA entries) [12, Lemma 5.15] Let F be a field, $f, g, r, s, t \in F[x]$ with $r = sf + tg,\ t \neq 0$, $\deg f > 0$, and

$$\deg r + \deg t < \deg f. \tag{3.3}$$

Moreover, let r_i, s_i, t_i for $0 \leq i \leq l+1$ be the elements of the ith row in the Extended Euclidean Algorithm for f and g. There exists a nonzero element $\alpha \in F[x]$ such that

$$r = \alpha r_j,\ s = \alpha s_j,\ t = \alpha t_j,$$

where $\deg r_j \leq \deg r < \deg r_{j-1}$.

Proof. By (3.3) we have $\deg r < \deg f = \deg r_0$, so there exists a row namely j in the Extended Euclidean Algorithm for inputs f and g, where $\deg r_j \leq \deg r < \deg r_{j-1}$. We have $r_j = s_j f + t_j g$ by Lemma 2.4 (vi), thus we obtain

$$tr_j - t_j r = (ts_j - t_j s)f. \tag{3.4}$$

Assume $ts_j \neq t_j s$ then the degree of the right hand side of (3.4) is at least $\deg f$, while

$$\begin{aligned}\deg(tr_j - t_j r) &\leq \max\{\deg t + \deg r_j, \deg t_j + \deg r\}\\ &\leq \max\{\deg t + \deg r, \deg f - \deg r_{j-1} + \deg r\}\\ &< \deg f,\end{aligned}$$

hence we have a contradiction which implies that $ts_j = t_j s$ or $s_j | s$, by Lemma 2.4 (vii). We write $s = \alpha s_j$ where $\alpha \in F[x] \backslash \{0\}$, then $t = \alpha t_j$ and $r = sf + tg = \alpha r_j$. □

The above lemma implies that any linear combination $r = sf + tg$ of f and g, with r and t having small degrees, is a multiple of a row in the Extended Euclidean Algorithm for inputs f an g. In the RFR problem, we are looking for a rational function n/d where $n/d \equiv g \mod f$ and $\deg n + \deg d < \deg f$. Thus according to Lemma 3.2 any solution for n and d is a multiple of some row in the EEA for inputs f and g.

3.2.1 Wang's Algorithm

Let F be a field, in the rational reconstruction problem we are looking for a rational function $n/d \in F[x]$ where $n/d \equiv g \mod f$, with $f, g \in F[x]$ and $\deg f > \deg g \geq 0$. A solution to the rational number reconstruction problem was first introduced by Wang in [1], however he

gave no proof for his algorithm. Afterwards in [2], Wang, Guy & Davenport showed that if a solution exists to the rational reconstruction problem, this solution is produced by Wang's algorithm. Moreover, they claimed that if the pair n, d is output by the algorithm then n/d is the expected solution. While there were some cases with no solution but Wang's Algorithm did not FAIL on them. This problem occurred because of not checking whether the condition $\gcd(d, f) = 1$ is met or not. Wang rectified this problem later in [3]. Algorithm 3.1 is an extension of Wang's algorithm for $F[x]$.

Algorithm 3.1: Wang's Rational Function Reconstruction Algorithm

---

Input: $f, g \in F[x]$ with F a field and $M = \deg f > \deg g \geq 0$.
Output: Either $n, d \in F[x]$ with $\deg n + \deg d < \deg f$, $\mathrm{lc}(d) = 1$, $\gcd(n, d) = 1$, $\gcd(f, d) = 1$ and $n/d \equiv g \mod f$, or FAIL implying no such n/d exists.

1. $N = \lfloor M/2 \rfloor,\ D = M - N - 1$
   $r_0 = f,\ t_0 = 0$
   $r_1 = g,\ t_1 = 1$

2. while $\deg r_1 > N$ do
   $q = r_0 \text{ quo } r_1$
   $(r_0, r_1) = (r_1, r_0 - qr_1)$
   $(t_0, t_1) = (t_1, t_0 - qt_1)$

3. if $\gcd(r_1, t_1) \neq 1$ then return FAIL. /∗ $\gcd(r_1, t_1) = \gcd(f, t_1)$ ∗/
   return $(r_1/\,\mathrm{lc}(t_1), t_1/\,\mathrm{lc}(t_1))$ /∗ $\deg t_1 = M - \deg r_0 < M - N = D + 1$ ∗/

Wang's algorithm outputs the rational function n/d if $\deg n \leq \lfloor \deg f/2 \rfloor$ and $\deg d \leq \lceil \deg f/2 \rceil - 1$, i.e. $\deg f \geq 2 \max(\deg n, \deg d)$. In step 3 of Algorithm 3.1 we have

$$r_1 = s_1 f + t_1 g \equiv t_1 g \mod f,$$

and $\deg t_1 = \deg f - \deg r_0 < M - N = D + 1$ or $\deg t_1 \leq D$. Thus if $\gcd(f, t_1) = 1$ then $(r_1/\,\mathrm{lc}(t_1))/(t_1/\,\mathrm{lc}(t_1))$ is a canonical form solution to the RFR problem. Collins and Encarnation in [18] point out that it is more efficient to make the test $\gcd(n, d) = 1$ instead of $\gcd(f, d) = 1$. By Lemma 2.4 (viii) we have $\gcd(n, d) = \gcd(d, f)$, thus instead of checking the invertibility of the denominator, we can check the coprimality of the numerator and the

denominator. This costs less since the size of n is strictly smaller than the size of f.

Cost Analysis of Wang's Algorithm

In Section 2.2.1 we showed that the cost of the EEA on inputs of size n is $O(n^2)$. Therefore, step 2 of Algorithm 3.1 costs $O(M^2)$. In step 3 we have one inversion and $O(M)$ multiplications in F. Computing $\gcd(r_1, t_1)$ takes another $O(M^2)$, thus the total cost of Wang's Algorithm is $O(M^2)$.

Remark 3.3. Given degree bound N, the FEEA can be used for returning r_j, a remainder in the remainder sequence of the EEA for monic inputs r_0 and r_1, satisfying $\deg r_j \leq N < \deg r_{j-1}$. Let $h = \eta_{r_0,r_1}(k)$, then in the FEEA with inputs r_0, r_1 and k we have

$$\deg r_{h+1} = \deg r_0 - \sum_{i=1}^{h+1} \deg q_i < \deg r_0 - k \leq \deg r_0 - \sum_{i=1}^{h} \deg q_i = \deg r_h,$$

according to (2.11), or equivalently,

$$\deg r_{h+1} \leq \deg r_0 - k - 1 < \deg r_h.$$

Therefore r_j is returned if we call the FEEA with inputs r_0, r_1 and $(\deg r_0 - N - 1)$.

Thus if we use the FEEA in steps 2 and 3 of Algorithm 3.1 then the total cost of Wang's algorithm would be of $O(\mathsf{M}(M) \log M)$ operations in F.

3.2.2 Maximal Quotient Rational Function Reconstruction

Wang's algorithm works well when the numerator and the denominator are both of almost the same degree, but in practice the degrees of the numerator and the denominator of the rational functions are not necessarily the same. For example if we want to recover the rational function $x/(x^5 + 1)$, Wang's algorithm needs the modulus f to be at least of degree 11, however the minimum number of points necessary for recovering the same rational function is 7. Since the degrees of the numerator and the denominator of the rational function are not always known, we do not know the best choice for N and D in advance. One approach could be to choose the rational function with the minimum total degree (numerator degree plus denominator degree).

Example 3.4. Let $F = \mathbb{Z}_{17}$, $f = \prod_{i=1}^{12}(x - i)$, $g = 6x^{11} + 13x^{10} + 7x^9 + 11x^8 + x^7 + 10x^6 + 15x^5 + x^4 + 13x^3 + 6x^2 + 3$. The Extended Euclidean Algorithm for f and g yields the following table.

i	$\deg q_i$	$\deg r_i$	$\deg t_i$	$\deg r_i + \deg t_i$
1	1	11	0	11
2	4	7	1	8
3	1	6	5	11
4	1	5	6	11
5	1	4	7	11
6	1	3	8	11
7	1	2	9	11
8	1	1	10	11

As illustrated in the table, r_2/t_2 has the minimum total degree of 8. Note that r_2/t_2 also corresponds to the quotient of maximal degree q_2. The reason for this is easily explained by the following lemma.

Lemma 3.5. Let F be a field and $f, g \in F[x]$. In the EEA for f and g we have

$$\deg r_i + \deg t_i + \deg q_i = \deg f$$

for $1 \le i \le l$ (l is the total number of steps of the EEA).

Proof. According to Lemma 2.4 (vii) we have $\deg t_i = m - \deg r_{i-1}$, thus

$$\deg r_i + \deg t_i + \deg q_i = \deg r_i + (m - \deg r_{i-1}) + \deg r_{i-1} - \deg r_i = m.$$

□

In [11], Monagan suggests a new method called *Maximal Quotient Rational Reconstruction* for reconstructing a rational number from its integer image modulo another integer number. Our algorithm for recovering rational functions is based on his method and is called Maximal Quotient Rational Function Reconstruction (MQRFR).

Let F be a field, $f, g \in F[x]$ with $\deg f > \deg g \ge 0$. We want to find a rational function $n/d \in F(x)$, where

$$n/d \equiv g \mod f, \quad \gcd(f, d) = 1, \quad \gcd(n, d) = 1, \quad \mathrm{lc}(d) = 1.$$

Let l denote the total number of steps of the EEA for f and g. The maximal quotient algorithm outputs a rational function $n/d = r_i/t_i$ with $\deg r_i + \deg t_i$ minimal for $i = 1 \ldots l$.

To speed up the algorithm we prefer to use the FEEA instead of the EEA. But as explained in previous chapter, the FEEA does not compute the intermediate remainders (r_i's). Thus we can not determine which pair of (r_i, t_i) we should choose. Lemma 3.5 resolves this problem.

Although the FEEA does not compute the intermediate remainders, it does compute all q_i's!! Also s_i and t_i are available as the entries of the first row of R_i. So according to Lemma 3.5 instead of finding the minimal $\deg r_i + \deg t_i$ we can find q_i, the quotient with maximal degree, using the FEEA. r_i is then obtained from s_i and t_i using two long multiplications ($r_i = s_i f + t_i g$).

The modified FEEA is called MQFEEA and is presented by Algorithm 3.2. In addition to the outputs returned by Algorithm 2.5, this algorithm returns three other values $q_{\max}, s_{\max}, t_{\max}$. The value of $q_{\max}$ is the quotient with maximal degree and $s_{\max}, t_{\max}$ represent the corresponding values of s and t that are in the same row with $q_{\max}$.

In step 3, after returning from the recursive call, $q_{\max}$ holds the quotient with maximal degree, between the first $j-1$ computed quotients. We have

$$R_j = \begin{pmatrix} s_j & t_j \\ s_{j+1} & t_{j+1} \end{pmatrix},$$

thus in step 6, if $\deg q_j > \deg q_{\max}$ we can easily update $s_{\max}$ and $t_{\max}$ by the entries of the first row of R_j. In step 8 $q_{\max}$ is updated by $q^*_{\max}$, if $\deg q^*_{\max} > \deg q_{\max}$. Let l represent the index of $q^*_{\max}$ in the EEA for r_0 and r_1. We need to compute s_l and t_l. We have

$$\begin{pmatrix} s_l & t_l \\ s_{l+1} & t_{l+1} \end{pmatrix} = R_l = Q_l Q_{l-1} \ldots Q_{j+1} R_j = \begin{pmatrix} s^*_{\max} & t^*_{\max} \\ M_{21} & M_{22} \end{pmatrix} R_j,$$

where $M_{21}, M_{22} \in F[x]$, hence

$$\begin{pmatrix} s_l & t_l \end{pmatrix} = \begin{pmatrix} s^*_{\max} & t^*_{\max} \end{pmatrix} R_j.$$

So to update the values of $s_{\max}$ and $t_{\max}$ by s_l, respectively, and t_l, we multiply the vector $\begin{pmatrix} s^*_{\max} & t^*_{\max} \end{pmatrix}$ by matrix R_j.

Remark 3.6. In Algorithm 3.2, we are just using the *degree* of the maximal quotient, thus instead of returning $q_{\max}$ we could return $\deg q_{\max}$.

Algorithm 3.2: Modified FEEA to return the maximal quotient(MQFEEA)

/* underlined parts illustrate modifications made to Algorithm 2.5 (FEEA). */

Input: r_0, r_1 two *monic* polynomials in $F[x]$ with $n_0 = \deg r_0 > n_1 = \deg r_1 \geq 0$ and $k \in \mathbb{N}$ with $n_0/2 \leq k \leq n_0$. /* n_0 is strictly greater than n_1. */

Output: $h = \eta_{r_0,r_1}(k) \in \mathbb{N}$, $\rho_{h+1} \in F$, $R_h = \begin{pmatrix} s_h & t_h \\ s_{h+1} & t_{h+1} \end{pmatrix}$ and $\begin{pmatrix} r_h \\ r_{h+1} \end{pmatrix} = R_h \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$, $\underline{q_{\max}, s_{\max}, t_{\max}}$.

1. if $r_1 = 0$ or $k < n_0 - n_1$ then
   return $0, 1, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}, \underline{1, 1, 0}$
   else if $n_0 <$ *cutoff* then /* *cutoff* degree of the FEEA */
   return EEA(r_0, r_1, k) /* EEA is modified to return $q_{\max}$, $s_{\max}$, $t_{\max}$ */

2. $k_1 = \lfloor k/2 \rfloor$

3. $r_0^* = r_0 \upharpoonright 2k_1$, $\quad r_1^* = r_1 \upharpoonright (2k_1 - (n_0 - n_1))$
   call the algorithm recursively by writing MQFEEA(r_0^*, r_1^*,k_1), to obtain
   $j - 1 = \eta_{r_0^*, r_1^*}(k_1)$, ρ_j^*, $R_{j-1}^* = Q_{j-1}^* Q_{j-2} \ldots Q_1$ where $Q_{j-1}^* = \begin{pmatrix} 0 & 1 \\ \frac{1}{\rho_j^*} & \frac{-q_{j-1}}{\rho_j^*} \end{pmatrix}$,
   $\begin{pmatrix} r_{j-1}^* \\ r_j^* \end{pmatrix} = R_{j-1}^* \begin{pmatrix} r_0^* \\ r_1^* \end{pmatrix}$ and $\underline{q_{\max}, s_{\max}, t_{\max}}$

4. /* in this step we want to determine ρ_j, r_{j-1}, r_j and R_{j-1}. */
   $\begin{pmatrix} r_{j-1} \\ \tilde{r}_j \end{pmatrix} = R_{j-1}^* \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$
   $R_{j-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1/\operatorname{lc}(\tilde{r}_j) \end{pmatrix} R_{j-1}^*$
   $\rho_j = \rho_j^* \operatorname{lc}(\tilde{r}_j)$, $\quad r_j = \tilde{r}_j / \operatorname{lc}(\tilde{r}_j)$

5. $\begin{pmatrix} n_{j-1} \\ n_j \end{pmatrix} = \begin{pmatrix} \deg r_{j-1} \\ \deg r_j \end{pmatrix}$
   if $r_j = 0$ or $k < n_0 - n_j$ then
   return $j - 1$, ρ_j, R_{j-1}, $\begin{pmatrix} r_{j-1} \\ r_j \end{pmatrix}$, $\underline{q_{\max}, s_{\max}, t_{\max}}$

---

6. $q_j = r_{j-1} \text{ quo } r_j$
   $\rho_{j+1} = \text{lc}(r_{j-1} - q_j r_j)$ /* for consistency we let $\text{lc}(0) = 1$. */
   $r_{j+1} = (r_{j-1} - q_j r_j)/\rho_{j+1}$
   $n_{j+1} = \deg r_{j+1}$
   $R_j = \begin{pmatrix} 0 & 1 \\ 1/\rho_{j+1} & -q_j/\rho_{j+1} \end{pmatrix} R_{j-1}$
   <u>if $\deg q_j > \deg q_{\max}$ then</u>
   <u>$q_{\max}, s_{\max}, t_{\max}{=}q_j, R_j[1,1], R_j[1,2]$</u>

7. $k_2 = k - (n_0 - n_j)$

8. $r_j^* = r_j \upharpoonright 2k_2, \quad r_{j+1}^* = r_{j+1} \upharpoonright (2k_2 - (n_j - n_{j+1}))$
   call the algorithm recursively by writing MQFEEA(r_j^*, r_{j+1}^*, k_2), to obtain
   $h - j = \eta_{r_j^*, r_{j+1}^*}(k_2)$, ρ_{h+1}^*, $\tilde{S} = Q_h^* Q_{h-1} \ldots Q_{j+1}$ where $Q_h^* = \begin{pmatrix} 0 & 1 \\ \frac{1}{\rho_{h+1}^*} & \frac{-q_h}{\rho_{h+1}^*} \end{pmatrix}$,
   $\begin{pmatrix} r_h^* \\ r_{h+1}^* \end{pmatrix} = \tilde{S} \begin{pmatrix} r_j^* \\ r_{j+1}^* \end{pmatrix}$, <u>$q_{\max}^*$, $s_{\max}^*$, $t_{\max}^*$</u>
   <u>if $\deg q_{\max}^* > \deg q_{\max}$ then</u>
   <u>$q_{\max} = q_{\max}^*$</u>
   <u>$\begin{pmatrix} s_{\max} & t_{\max} \end{pmatrix} = \begin{pmatrix} s_{\max}^* & t_{\max}^* \end{pmatrix} R_j$</u>

9. $\begin{pmatrix} r_h \\ \tilde{r}_{h+1} \end{pmatrix} = \tilde{S} \begin{pmatrix} r_j \\ r_{j+1} \end{pmatrix}$
   $S = \begin{pmatrix} 1 & 0 \\ 0 & 1/\text{lc}(\tilde{r}_{h+1}) \end{pmatrix} \tilde{S}$
   $\rho_{h+1} = \rho_{h+1}^* \text{lc}(\tilde{r}_{h+1}), \quad r_{h+1} = \tilde{r}_{h+1}/\text{lc}(\tilde{r}_{h+1})$

10. return h, ρ_{h+1}, SR_j, $\begin{pmatrix} r_h \\ r_{h+1} \end{pmatrix}$, <u>$q_{\max}$, $s_{\max}$, $t_{\max}$</u>.

Let $F = \mathbb{Z}_p$, p a prime, $f, g \in F[x]$ and $\deg f > \deg g \geq 0$. We want to recover $n/d \in F[x]$ using the maximal quotient algorithm where $n/d \equiv g \mod f$, $\gcd(n, d) = 1$ and $\gcd(f, d) = 1$. According to Lemma 3.2 the solution to this problem is the pair (r_j, t_j), where r_j and t_j are the elements of the jth row in the EEA for inputs f and g. If $\deg f > 2(\deg n + \deg d)$, then we have $\deg q_j > \deg f/2$ and thus q_j is the unique maximal quotient. This implies that by imposing $\deg f > 2(\deg n + \deg d)$, the expected rational function is returned with probability 1.

The following conjecture implies that if we impose $\deg f > \deg n + \deg d + 1$ or, equivalently, we require $\deg q_j > 1$, then the probability of getting a correct result is still high, provided that p is not small compared to $\deg f$.

Conjecture 3.7. Let $F = \mathbb{Z}_p$, where p is prime. Let $f, g \in F[x]$ where $f = \prod_{i=1}^{n}(x - \alpha_i)$ and $n = \deg f > \deg g \geq 0$. Let q be a quotient in the EEA for inputs f, g and $k \in \mathbb{N}\backslash\{1\}$. If $\alpha_i \in F$ is chosen uniformally at random and g is a random polynomial, then

$$\text{Prob}(\deg q \geq k) \simeq \frac{n}{p^{k-1}}.$$

We run the EEA with inputs f and a randomly chosen polynomial g. Our conjecture is that the number of polynomials g for which there is a quotient of degree at least k in the EEA is bounded by $(n - k + 1)p^{n-k+1}$. The total number of possible choices of g is $p^n - 1$. Thus

$$\text{Prob}(\deg q \geq k) = \frac{(n - k + 1)p^{n-k+1}}{p^n - 1} \simeq \frac{n - k + 1}{p^{k-1}}.$$

The maximal quotient algorithm is presented by Algorithm 3.3. It is supposed to return the rational function $n/d = r_i/t_i$ where q_i is the quotient with the maximal degree. Let r, t and $q_{\max}$ be the elements of the same row of the EEA with f and g as input. In step 3 we have

$$\frac{r}{t} = \frac{\frac{\tilde{s}}{\text{lc}(f)}f + \frac{\tilde{t}}{\text{lc}(g)}g}{\frac{\tilde{t}}{\text{lc}(g)}} = \frac{\text{lc}(g)(\tilde{s}r_0 + \tilde{t}r_1)}{\tilde{t}} = \text{lc}(g)\frac{\tilde{r}}{\tilde{t}},$$

and thus $\gcd(r, t) = \gcd(\tilde{r}, \tilde{t})$. Therefore if $\gcd(\tilde{r}, \tilde{t}) \neq 1$, then in step 4 $n = \text{lc}(g)/\text{lc}(\tilde{t})\tilde{r}$ and $d = \tilde{t}/\text{lc}(\tilde{t})$ is returned as the canonical solution.

ALGORITHM 3.3: Maximal Quotient Rational Function Reconstruction (MQRFR)

Input: $f, g \in \mathbb{Z}_p[x]$, where p is prime and $m = \deg f > \deg g \geq 0$.
Output: Either $n, d \in \mathbb{Z}_p[x]$ satisfying $\deg n + \deg d + 1 < \deg f,\ n/d \equiv g \mod f$, $\gcd(n, d) = \gcd(f, d) = 1, \operatorname{lc}(d) = 1$ or FAIL implying there is no such solution.

1. $r_0 = f/\operatorname{lc}(f), \quad r_1 = g/\operatorname{lc}(g)$
2. $h, \rho_{h+1}, R_h, \begin{pmatrix} r_h \\ r_{h+1} \end{pmatrix}, q_{\max}, \tilde{s}, \tilde{t} = \text{MQFEEA}(r_0, r_1, m)$
   if $\deg q_{\max} \leq 1$ then
     return FAIL
3. $\tilde{r} = \tilde{s}r_0 + \tilde{t}r_1$
   if $\gcd(\tilde{r}, \tilde{t}) \neq 1$ then
     return FAIL
4. $n = \operatorname{lc}(g)/\operatorname{lc}(\tilde{t}).\tilde{r}$
   $d = \tilde{t}/\operatorname{lc}(\tilde{t})$
   return (n, d).

Cost Analysis of the MQRFR

As mentioned before, Algorithm 3.2 is a modification of Algorithm 2.5. Among all the modifications made only the multiplication in step 8 might affect the asymptotic cost of the algorithm which originally was $O(\mathsf{M}(k)\log k)$ for input k. The entries of matrix R_j are at most of degree $n_0 - n_j \leq k$, moreover by Lemma 2.4 (ix), $\deg s^*_{\max} < n_{j+1} < 2k$ and $\deg t^*_{\max} < n_j < 2k$. Thus multiplying $\begin{pmatrix} s^*_{\max} & t^*_{\max} \end{pmatrix}$ by R_j at most takes $8\mathsf{M}(k) + O(k)$ operations in F and does not change the asymptotic cost of the algorithm.

Step 1 of Algorithm 3.3 consists of two inversions in $\mathbb{Z}_p$ and two multiplications of $O(m)$ in $\mathbb{Z}_p$. Step 2 costs $O(\mathsf{M}(m)\log m)$. We have $\deg s < \deg g < m$ and $\deg t < \deg f = m$. Thus to compute r in step 3, we perform two multiplications on polynomials of size at most m and one addition that costs $O(2m)$. The total cost for computing r is thus $2\mathsf{M}(m) + O(m)$. r is a remainder in the remainder sequence generated by the Euclidean Algorithm for f and g, thus we have $\deg r < \deg f = m$. Checking the coprimality of r and t, using the

FEEA, takes $O(\mathsf{M}(m)\log m)$ operations in F. In step 4 we have one inversion and at most $2m$ multiplications in $\mathbb{Z}_p$ that costs $O(m)$. Thus the total cost of Algorithm 3.3 is $O(\mathsf{M}(m)\log m)$ operations in $\mathbb{Z}_p$.

Chapter 4

Polynomial GCD Computation

In this chapter we will explain the application of Rational Function Reconstruction in computing the GCD of multivariate polynomials. We have modified Brown's algorithm to use the maximal quotient algorithm. Our modification reduces the number of evaluation points needed by the algorithm.

4.1 Multivariate GCD Computation (Brown's Algorithm)

Definition 4.1. Let R_1 and R_2 be two rings. The mapping $\phi : R_1 \to R_2$ is a *ring morphism* or a *homomorphism* if

(i) $\phi(a+b) = \phi(a) + \phi(b)$ for all $a, b \in R_1$,

(ii) $\phi(ab) = \phi(a)\phi(b)$ for all $a, b \in R_1$,

(iii) $\phi(1) = 1$.

Brown's algorithm applies the following homomorphisms:

- The *modular homomorphism* $\phi_m : \mathbb{Z}[x_1, \ldots, x_k] \to \mathbb{Z}_m[x_1, \ldots, x_k]$ that replaces all the integer coefficients of a polynomial $f \in \mathbb{Z}[x_1, \ldots, x_k]$ by their modulo m representation.
- The *evaluation homomorphism* $\phi_{x_i - \alpha} : D[x_1, \ldots, x_k] \to D[x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k]$ that substitutes the value of $\alpha \in D$ for the indeterminate x_i in the polynomial $f \in D[x_1, \ldots, x_k]$.

Let $f \in \mathbb{Z}[x_1, \ldots, x_k]$. For each term in f, we define a vector of size k whose ith element is the power of x_i in that term. Degree of f with respect to $x_1, \ldots, x_k$ (or simply degree of f), which is denoted by $\deg_{[x_1,\ldots,x_k]} f$, is defined to be the maximum of these vectors when compared lexicographically.

The leading coefficient of a multivariate polynomial $f \in \mathbb{Z}[x_1, \ldots, x_k]$ with respect to $x_1, \ldots, x_i$ $(i \le k)$ or $\mathrm{lc}_{[x_1,\ldots,x_i]}(f)$ is defined to be the coefficient of the term with the highest degree with respect to $x_1, \ldots, x_i$.

Definition 4.2. Let $f, g \in \mathbb{Z}[x_1, \ldots, x_k]$, $h = \gcd(f, g)$ and p be a prime.

- If $\mathrm{lc}_{[x_1,\ldots,x_k]}(h)$ disappears mod p then we call p a *bad prime*.
- Let $f_p = f \mod p$, $g_p = g \mod p$ and h_p be the output of the Euclidean Algorithm mod p on f_p and g_p. If $\deg_{[x_1,\ldots,x_k]} h_p$ is higher than $\deg_{[x_1,\ldots,x_k]} h$, then p is called an *unlucky prime*.

Example 4.3. Let $f = (x + 7y)(5xy + 1)$ and $g = x(5xy + 1)$. Then $h = 5xy + 1$, $\gcd(f_5, g_5) = 1$ and $\gcd(f_7, g_7) = 5x^2y + x$, and thus $p = 5$ is a bad prime and $p = 7$ is an unlucky prime.

Definition 4.4. Let $f, g \in \mathbb{Z}_p[x_1, \ldots, x_k]$, $h = \gcd(f, g) \mod p$ and $\alpha \in \mathbb{Z}_p$, where p is a prime.

- If the leading coefficient of h disappears mod $x_k - \alpha$, then we call α a *bad evaluation point*.
- Let $f_{x_k-\alpha} = f \mod (x_k - \alpha)$, $g_{x_k-\alpha} = g \mod (x_k - \alpha)$ and $h_{x_k-\alpha}$ be the output of the Euclidean Algorithm mod p on $f_{x_k-\alpha}$ and $g_{x_k-\alpha}$. If $\deg_{[x_1,\ldots,x_{k-1}]} h_{x_k-\alpha}$ is higher than $\deg_{[x_1,\ldots,x_{k-1}]} h$, then α is called an *unlucky evaluation point*.

Example 4.5. Let $f = ((y - 1)x + 1)(x - 2)$ and $g = ((y - 1)x + 1)(x - y)$. Then $h = (y - 1)x + 1$, $\gcd(f_{y-1}, g_{y-1}) = 1$ and $\gcd(f_{y-2}, g_{y-2}) = (x + 1)(x - 2)$, and thus $y = 1$ is a bad evaluation point and $y = 2$ is an unlucky evaluation point.

Assume f and g are two multivariate polynomials as defined above and $h = \gcd(f, g)$. Let ϕ be a modular or an evaluation homomorphism. It is obvious that $\mathrm{lc}(h) | \mathrm{lc}(f)$ and $\mathrm{lc}(h) | \mathrm{lc}(g)$, which implies that $\mathrm{lc}(h) | \gcd(\mathrm{lc}(f), \mathrm{lc}(g))$. Thus

$$\phi(\gcd(\mathrm{lc}(f), \mathrm{lc}(g))) \neq 0 \Longrightarrow \phi(\mathrm{lc}(h)) \neq 0. \tag{4.1}$$

Relation (4.1) can be useful for avoiding bad primes and bad evaluation points when computing the GCD of two multivariate polynomials. The following lemma helps us to detect and discard unlucky primes and evaluation points.

Lemma 4.6. Let $\phi : R \to R'$ be a homomorphism of rings, $f, g \in R[x_1, \ldots, x_k]$ and $h = \gcd(f, g)$. Let $f = \bar{f}h$ and $g = \bar{g}h$. Assume $\phi(\mathrm{lc}(h)) \neq 0$ and at least one of $\phi(\bar{f})$ and $\phi(\bar{g})$ is nonzero. Then

$$\deg \gcd(\phi(f), \phi(g)) \geq \deg \gcd(f, g).$$

Proof. By Definition 4.1 (ii) we have

$$\phi(f) = \phi(\bar{f})\phi(h), \phi(g) = \phi(\bar{g})\phi(h),$$

and thus we obtain

$$\deg \gcd(\phi(f), \phi(g)) \geq \deg \phi(h) = \deg \gcd(f, g),$$

since we assumed $\phi(\mathrm{lc}(h)) \neq 0$. □

Example 4.7. Let f and g be defined as in Example 4.5. We have $h = \gcd(f, g) = (y-1)x + 1$. If $y = 2$ is chosen to be an evaluation point then we have $h_{y-2} = x + 1$ and $\gcd(f_{y-2}, g_{y-2}) = (x+1)(x-2)$, that is, $\deg \gcd(f_{y-2}, g_{y-2}) > \deg h_{y-2}$. Thus $y = 2$ is an unlucky evaluation point and should be discarded.

Assume for homomorphism ϕ we have $\gcd(\phi(f), \phi(g)) = 1$ and $\phi(\mathrm{lc}(\gcd(f, g))) \neq 0$. Then according to Lemma 4.6 we have

$$\deg \gcd(\phi(f), \phi(g)) = 0 \geq \deg \gcd(f, g),$$

which implies that f, g are relatively prime. Thus another use of Lemma 4.6 is to help us detect the coprimality of input polynomials.

Originally Collins in [19] developed an algorithm for computing the GCD of univariate polynomials using homomorphic reductions. Then Brown in [20] extended the algorithm to compute the GCD in the multivariate case using homomorphisms. Brown's original algorithm did not use trial divisions. However, in [21] the algorithm was modified to use trial division and less evaluation points. In this thesis whenever we refer to Brown's algorithm we

refer to the modified one. Brown's algorithm is a composition of modular homomorphisms (MGCD algorithm) and evaluation homomorphisms (PGCD algorithm).

To avoid the problem of coefficient growth, MGCD gets two multivariate polynomials $f, g \in \mathbb{Z}[x_1, \ldots, x_k]$ and applies the modular homomorphism $\phi_p : \mathbb{Z} \to \mathbb{Z}_p$ on the coefficients of f and g, with p a random machine prime, e.g. a 32 bit prime on a 32 bit machine. At the end of the algorithm the modular homomorphism is inverted by applying the Chinese Remainder Algorithm on homomorphic images.

The PGCD algorithm, presented by Algorithm 4.1, gets polynomials $f, g \in \mathbb{Z}_p[x_1, \ldots, x_k]$ and outputs $h = \gcd(f, g) \in \mathbb{Z}_p[x_1, \ldots, x_k]$. It recursively makes use of evaluation homomorphism $\phi_{x_k - \alpha} : \mathbb{Z}_p[x_1, \ldots, x_k] \to \mathbb{Z}_p[x_1, \ldots, x_{k-1}]$ to ultimately get to the Euclidean domain $\mathbb{Z}_p[x_1]$ where the ordinary Euclidean algorithm can be used. In order to be able to recover the solution in the original domain we need more than one projection. The evaluation homomorphism is inverted by interpolating homomorphic images. Since MGCD and PGCD algorithms are very similar, and in the next section we are going to modify PGCD so that it uses Rational Function Reconstruction, we have chosen to present only the PGCD algorithm.

As presented in Algorithm 4.1, to compute $\gcd(f, g)$ PGCD computes the *contents*[*] and the *primitive parts*[†] of f and g, and then computes the GCD using

$$\gcd(f, g) = \gcd(\text{cont}(f), \text{cont}(g)) \gcd(\text{pp}(f), \text{pp}(g)).$$

PGCD always returns a monic GCD, thus $v_{x_k - \alpha}$ is always monic, although $\gcd(f, g)$ is not necessarily monic. To recover the correct GCD, in Algorithm 4.1, $v_{x_k - \alpha}$ is multiplied by $\gamma(\alpha)$ (leading coefficient correction).

One way of determining whether we should stop constructing images or not is to test whether $h = \text{pp}(u)$ divides f and g or not. But these divisions are expensive and it is better to avoid them as much as possible. One simple way to avoid attempting the divisions is to check whether $\text{lc}(u)$ is equal to γ or not and if it is then do the divisions. A better solution is to divide h into f and g only when the result of interpolation does not change in two consecutive iterations.

---

[*]The content of a nonzero polynomial is the unit normal GCD of its coefficients.

[†]The primitive part $\text{pp}(f)$ of f is defined by $f = \text{cont}(f) \cdot \text{pp}(f)$.

ALGORITHM 4.1: Brown's multivariate reduction algorithm (PGCD)

Input: $f, g \in \mathbb{Z}_p[x_1, \ldots, x_k]$
Output: $\gcd(f, g) \in \mathbb{Z}_p[x_1, \ldots, x_k]$

1. if $k = 1$ then return $\gcd(f, g)$. /* use FEEA to compute $\gcd(f, g) \in \mathbb{Z}_p[x_1]$ */

2. $c_f = \mathrm{cont}_{[x_1,\ldots,x_{k-1}]}(f)$, $c_g = \mathrm{cont}_{[x_1,\ldots,x_{k-1}]}(g)$ /* $c_f, c_g \in \mathbb{Z}_p[x_k]$ */
   $f = f/c_f$, $g = g/c_g$ /* f, g are now primitive */
   $l_f = \mathrm{lc}_{[x_1,\ldots,x_{k-1}]}(f)$, $l_g = \mathrm{lc}_{[x_1,\ldots,x_{k-1}]}(g)$ /* $l_f, l_g \in \mathbb{Z}_p[x_k]$ */
   $\mathbf{d}_f = \deg_{[x_1,\ldots,x_{k-1}]} f$, $\mathbf{d}_g = \deg_{[x_1,\ldots,x_{k-1}]} g$ /* $\mathbf{d}_f, \mathbf{d}_g$ are vector degrees */

3. $c_h = \gcd(c_f, c_g)$ /* $c_h \in \mathbb{Z}_p[x_k]$ and holds the content of the output */
   $\gamma = \gcd(l_f, l_g)$ /* $\gamma \in \mathbb{Z}_p[x_k]$ */
   $\mathbf{n} = \min(\mathbf{d}_f, \mathbf{d}_g)$ /* $\mathbf{n}$ is a vector holding the minimum of d_f and d_g */
   $(m, u) = (1, 1)$

4. while true do
   α = a new random element of $\mathbb{Z}_p$, such that $\gamma(\alpha) \neq 0$
   $f_{x_k-\alpha} = f \mod (x_k - \alpha)$
   $g_{x_k-\alpha} = g \mod (x_k - \alpha)$ /* $f_{x_k-\alpha}, g_{x_k-\alpha} \in \mathbb{Z}_p[x_1, \ldots, x_{k-1}]$ */
   $v_{x_k-\alpha} = \mathrm{PGCD}(f_{x_k-\alpha}, g_{x_k-\alpha}, p)$ /* $v_{x_k-\alpha} \in \mathbb{Z}_p[x_1, \ldots, x_{k-1}]$ */
   if $v_{x_k-\alpha} = 1$ then return c_h
   $u_{x_k-\alpha} = \gamma(\alpha) v_{x_k-\alpha}$ /* solve the leading coefficient problem */ (i)

   $\mathbf{d} = \deg_{[x_1,\ldots,x_{k-1}]} u_{x_k-\alpha}$ /* $\mathbf{d}$ is a vector degree */
   if $\mathbf{d} > \mathbf{n}$ then /* skip this (unlucky) evaluation point */
   else if $\mathbf{d} < \mathbf{n}$ then /* previous points were unlucky */
      $(m, u) = (x_k - \alpha, u_{x_k-\alpha})$
      $\mathbf{n} = \mathbf{d}$
   else
      $(m, u) = ((x_k - \alpha)m, \mathrm{Interp}(m, u, \alpha, u_{x_k-\alpha}))$ /* $u \in \mathbb{Z}_p[x_1, \ldots, x_k]$ */

      if $\mathrm{lc}_{[x_1,\ldots,x_{k-1}]}(u) = \gamma$ then (ii)
         $h = \mathrm{pp}_{[x_1,\ldots,x_{k-1}]}(u)$ (iii)
         if $h|f$ and $h|g$ then return $c_h h$ (iv)

The following example shows how the GCD of two bivariate polynomials is computed using the PGCD algorithm.

Example 4.8. Let $f = (yx^2 + yx + 1)((y^2 + 1)x + 1)$, $g = (yx^2 + yx + 1)((y^2 + 1)x + 2))$ and $p = 7$. We have $\gamma = y^3 + y$. Let us assume α is initialized by 1 and is increased by 1 in each iteration. The following table shows the value of $u_{x_k-\alpha}$ and u in each iteration.

α	$\gamma(\alpha)$	$u_{x_k-\alpha}$	u
1	2	$2x^2 + 2x + 2$	$2x^2 + 2x + 2$
2	3	$3x^2 + 3x + 5$	$(y + 1)x^2 + (y + 1)x + 3y + 6$
3	2	$2x^2 + 2x + 3$	$(6y^2 + 4y + 6)x^2 + (6y^2 + 4y + 6)x + y^2 + 1$
4	5	$5x^2 + 5x + 3$	$(y^3 + y)x^2 + (y^3 + y)x + y^2 + 1$

At the end of the fourth iteration $\mathrm{lc}_y(u) = \gamma$. Thus $h = pp(u) = yx^2 + yx + 1$ is returned after making sure that it divides both f and g.

4.2 Application of RFR to Brown's Algorithm

As mentioned earlier, to solve the leading coefficient problem, in Brown's PGCD algorithm the homomorphic images are multiplied by the GCD of the leading coefficients of input polynomials. Another solution to the leading coefficient problem is to use the Rational Function Reconstruction algorithm. Originally Encarnación in [9] used Wang's rational number reconstruction algorithm for computing the GCD of univariate polynomials over algebraic number fields.

We have marked two rows of PGCD which should be modified for this purpose. Row (i) should be deleted, $v_{x_k-\alpha}$ should be replaced by $u_{x_k-\alpha}$ and rows (ii), (iii) and (iv) should be replaced by the following code

---

$\hat{h} = \mathrm{LCR}(m, u)$
if $\hat{h} \neq$ FAIL then
    let h represent the result of clearing denominators of $\hat{h}$
    if $h|f$ and $h|g$ then return $c_h h$

Algorithm 4.2 displays the body of LCR. This algorithm gets a univariate polynomial m in $\mathbb{Z}_p[x_k]$ and a multivariate polynomial u in $\mathbb{Z}_p[x_1, \ldots, x_k]$ as input, and for each coefficient of u in $\mathbb{Z}_p[x_k]$ attempts to reconstruct a rational function in $\mathbb{Z}_p(x_k)$, resulting a polynomial in $\mathbb{Z}_p(x_k)[x_1, \ldots, x_{k-1}]$.

ALGORITHM 4.2: Retrieval of the leading coefficient (LCR)

Input: $m \in \mathbb{Z}_p[x_k]$, $u \in \mathbb{Z}_p[x_1, \ldots, x_k]$,
Output: $h \in \mathbb{Z}_p[x_1, \ldots, x_k]$ where $h \equiv u \mod m$.

1. if deg $m = 1$ then return FAIL /∗ $\deg q < 2$ and thus MQRFR fails ∗/

2. while true do
   - $r =$ MQRFR(m, nextcoeff(u) $\in \mathbb{Z}_p[x_k]$)
   - if $r =$ FAIL then
     - return FAIL
   - else /∗ r is a rational function in $\mathbb{Z}_p[x_k]$ ∗/
   - replace current coefficient of u by r

   if MQRFR did not fail on any of the coefficients then return u.

Example 4.9. Let f, g and p be defined as in Example 4.8. If we use LCR in the PGCD algorithm then we will have

α	$u_{x_k-\alpha}$	u	$\hat{h}$
1	x^2+x+1	x^2+x+1	FAIL
2	x^2+x+4	$x^2+x+3y+5$	FAIL
3	x^2+x+5	x^2+x+6y^2+6y+3	x^2+x+1/y

As illustrated in the above table, we need only 3 points, while in Example 4.8, 4 points were required. After clearing the denominators we get $h = yx^2 + yx + 1$ which is the same result as what we had obtained before.

In practice if MQRFR does not fail on one coefficient of u in one call, with high probability it will not fail on the same coefficient in subsequent calls either. Thus we can reduce the total number of times MQRFR is called by using a global variable to keep track of the index of the last coefficient of u on which MQRFR failed.

Example 4.10. Let $u = x^3+(y+1)x^2+(4y^3+2y+6)x+(2y^3+y+6)$, $m = y(y+1)(y+2)(y+3)$ and $p = 7$. The following table illustrates the intermediate values of u when LCR is called with m, u as inputs. At the end of the fourth iteration denominators of u are cleared and $u = (y+6)x^3 + (y^2+6)x^2 + (y+1)x + 1$ is returned.

nextcoeff(u)	r	u
1	1	$x^3 + (y+1)x^2 + (4y^3 + 2y + 6)x + (2y^3 + y + 6)$
$y+1$	$y+1$	$x^3 + (y+1)x^2 + (4y^3 + 2y + 6)x + (2y^3 + y + 6)$
$4y^3 + 2y + 6$	$\frac{y+1}{y+6}$	$x^3 + (y+1)x^2 + (\frac{y+1}{y+6})x + (2y^3 + y + 6)$
$2y^3 + y + 6$	$\frac{1}{y+6}$	$x^3 + (y+1)x^2 + (\frac{y+1}{y+6})x + (\frac{1}{y+6})$

Now want to obtain an upper bound for the number of evaluation points required for computing the primitive GCD of two multivariate polynomials using the modified PGCD algorithm. Let $f, g, H \in \mathbb{Z}_p[x_1, \ldots, x_k]$, where H is the primitive GCD of f and g in x_k. H can be expressed in the following form

$$H = c_n(x_k)t_n(\mathbf{x}) + c_{n-1}(x_k)t_{n-1}(\mathbf{x}) + \ldots + c_0(x_k)t_0(\mathbf{x}),$$

where $\mathbf{x} = x_1, \ldots, x_{k-1}$, $c_i(x_k)$ is a univariate polynomial in $\mathbb{Z}_p[x_k]$ and $t_i(\mathbf{x})$ is a monomial in indeterminants $x_1, \ldots, x_{k-1}$. We assume $\mathrm{lc}_{[x_1,\ldots,x_{k-1}]}(H) = c_n(x_k)$. Let d represent the minimum number of points required for recovering H using polynomial interpolation for x_k, that is

$$d = \deg_{x_k} H + 1 = \max_{0 \le i \le n} \{\deg c_i(x_k)\} + 1.$$

Before the last trial division in modified PGCD, which results in outputting the GCD, LCR is called on u and m where u is a monic multivariate polynomial in $\mathbb{Z}_p[x_1, \ldots, x_k]$ and $m = (x_k - \alpha_1)\ldots(x_k - \alpha_l)$ is a univariate polynomial in $\mathbb{Z}_p[x_k]$. Thus the last value of $\hat{h}$ =LCR(m, u) before returning H is

$$\begin{aligned}\hat{h} &= t_n(\mathbf{x}) + \frac{a_{n-1}(x_k)}{b_{n-1}(x_k)}t_{n-1}(\mathbf{x}) + \ldots + \frac{a_0(x_k)}{b_0(x_k)}t_0(\mathbf{x}) \\ &= t_n(\mathbf{x}) + \frac{c_{n-1}(x_k)}{c_n(x_k)}t_{n-1}(\mathbf{x}) + \ldots + \frac{c_0(x_k)}{c_n(x_k)}t_0(\mathbf{x}),\end{aligned}$$

where

$$c_n(x_k) = \mathrm{lcm}(b_{n-1}(x_k), \ldots, b_0(x_k)).$$

We have

$$a_i(x_k)|c_i(x_k),\ \deg c_i(x_k) < d \Longrightarrow \deg a_i(x_k) < d,$$

$$b_i(x_k)|c_n(x_k),\ \deg c_n(x_k) < d \Longrightarrow \deg b_i(x_k) < d,$$

for $0 \le i \le n-1$. As shown before, if

$$\deg m > 2(\max_{0\le i\le n-1}\{\deg a_i(x_k) + \deg b_i(x_k)\}),$$

then with probability 1, the modified PGCD algorithm returns $c_h H$. Thus in the worst case $4d$ evaluation points are required. But even if we have

$$\deg m \ge \max_{0\le i\le n-1}\{\deg a_i(x_k) + \deg b_i(x_k)\} + 2,$$

then with high probability $c_h H$ is resulted. Therefore in the normal case $2d$ evaluation points are required.

On the other hand, in Algorithm 4.1 we first compute $\frac{\gamma(x_k)}{c_n(x_k)}H$ and then take the primitive part by removing the common factor. Therefore the total number of evaluation points required to obtain H in Algorithm 4.1 is at least $(\deg \gamma(x_k) - \deg c_n(x_k) + d)$. This number may be much greater than d when $\gamma(x_k)$ is of a large degree and $\deg c_n(x_k) = 1$.

In the square-free factorization algorithm for a polynomial f we need to compute $\gcd(f, f')$. The following example shows the difference between the minimum number of evaluation points required in the original and modified PGCD algorithm when computing $\gcd(f, f')$.

Example 4.11. Let $f = (xy+1)^2(y^{100}x+1)$ and $g = f'_x = (xy+1)(3xy^{100}+y^{99}+2)y$. Then H, the primitive $\gcd(f, g)$ in y, is $xy + 1$. In this example $d = \deg_y H + 1 = 2, \deg\gamma = 102$ and $\deg_y \mathrm{lc}_x(H) = 1$. Thus the original PGCD algorithm requires at least 103 evaluation points while the modified one requires only 3 evaluation points to compute $\gcd(f, f')$.

As stated before we also try to minimize the number of times the trial divisions are attempted. Rational Function Reconstruction not only solves the leading coefficient problem but also helps us with reducing the number of trial division tests, so that with high probability the divisions are performed only on the true GCD. The point is that when LCR fails on a coefficient, it returns to PGCD and PGCD goes back to the beginning of the while loop without performing a division.

Let n be the number of coefficients of u with respect to $[x_1, \ldots, x_{k-1}]$ and l be the number of evaluation points required for computing the GCD. Then the expected number of times MQRFR is called is of $O(l+n)$. Note that as soon as m gets large enough MQRFR will not fail on any coefficient and exactly n more calls to MQRFR are made.

Chapter 5

Summary

We have designed and implemented a Fast Rational Function Reconstruction algorithm based on Monagan's Maximal Quotient Rational Number Reconstruction algorithm [11]. In contrast to Wang's algorithm, the maximal quotient algorithm does not require any degree bounds for the numerator and the denominator. Moreover, with high probability it requires only one more point than the minimum necessary to reconstruct the expected rational function.

The maximal quotient rational function reconstruction algorithm is based on the Extended Euclidean Algorithm. We have implemented the maximal quotient algorithm for polynomials in $\mathbb{Z}_p[x]$ where p is a machine prime. To speed up the reconstruction algorithm we have implemented Karatsuba's algorithm for polynomial multiplication over $\mathbb{Z}_p[x]$ and Schönhage's Fast Extended Euclidean Algorithm for $\mathbb{Z}_p[x]$. We followed the presentation of Schönhage's algorithm in [12]. The most difficult and time consuming part of the thesis was understanding the details of the FEEA.

To show one of the applications of this algorithm, we have modified Brown's modular GCD algorithm to use the maximal quotient algorithm. The modification reduces the number of evaluation points required by the algorithm. Also it reduces the number of times the trial divisions are attempted. The rational function reconstruction algorithm can be used to solve the leading coefficient problem when computing the GCD of two multivariate polynomials using Brown's algorithm.

Bibliography

[1] Paul S. Wang. A p-adic Algorithm for Univariate Partial Fractions. In *Proceedings of the fourth ACM Symposium on Symbolic and Algebraic Computation*, pages 212–217. ACM Press: New York, NY, 1981.

[2] P. S. Wang, M. J. T. Guy, and J. H. Davenport. p-adic reconstruction of rational numbers. *SIGSAM Bulletin*, 16(2), 1982.

[3] Paul S. Wang. Early detection of true factors in univariate polynomial factorization. In *EUROCAL '83: Proceedings of the European Computer Algebra Conference on Computer Algebra*, pages 225–235, London, UK, 1983. Springer-Verlag.

[4] George E. Collins and Mark J. Encarnación. Improved techniques for factoring univariate polynomials. *J. Symb. Comput.*, 21(3):313–327, 1996.

[5] Carlo Traverso. Gröbner trace algorithms. In *ISAAC '88: Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation*, pages 125–138, London, UK, 1989. Springer-Verlag.

[6] T. Sasaki and T. Takeshima. A modular method for Gröbner-basis construction over $\mathbb{Q}$ and solving system of algebraic equations. *J. Inf. Process.*, 12(4):371–379, 1989.

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

[8] Victor Y. Pan and Xinmao Wang. Acceleration of Euclidean algorithm and extensions. In *Proceedings of the International Conference on Symbolic and Algebraic Computation*, pages 207–213. ACM Press: New York, NY, 2002.

[9] Mark J. Encarnación. On a modular algorithm for computing gcds of polynomials over algebraic number fields. In *ISSAC '94: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 58–65. ACM Press: New York, NY, 1994.

[10] Mark van Hoeij and Michael Monagan. Algorithms for polynomial gcd computation over algebraic function fields. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 297–304. ACM Press: New York, NY, 2004.

[11] Michael Monagan. Maximal quotient rational reconstruction: An almost optimal algorithm for rational reconstruction. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 243–249. ACM Press: New York, NY, 2004.

[12] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press: Cambridge, New York, Port Melbourne, Madrid, Cape Town, second edition, 2003.

[13] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics—Doklady*, 7(7):595–596, January 1963.

[14] Roman E. Maeder. Storage allocation for the Karatsuba integer multiplication algorithm. In *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems, DISCO'93 (Gmunden, Austria, September 1993)*, volume 722 of *LNCS*, pages 59–65. Springer-Verlag, 1993.

[15] A. Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.

[16] R. T. Moenck. Fast computation of gcds. In *STOC '73: Proceedings of the fifth annual ACM Symposium on Theory of Computing*, pages 142–151. ACM Press: New York, NY, 1973.

[17] Peter Lawrence Montgomery. *An FFT extension of the elliptic curve method of factorization*. PhD thesis, Los Angeles, CA, USA, 1992.

[18] George E. Collins and Mark J. Encarnación. Efficient rational number reconstruction. *Journal of Symbolic Computation*, 20(3):287–297, 1995.

[19] George E. Collins. Subresultants and reduced polynomial remainder sequences. *J. ACM*, 14(1):128–142, 1967.

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

[21] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers: Boston/Dordrecht/London, 2002.