SPARSE MODULAR GCD ALGORITHM FOR POLYNOMIALS OVER ALGEBRAIC FUNCTION FIELDS

by

Seyed Mohammad Mahdi Javadi

B.Sc., Sharif University of Technology, 2004

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

SIMON FRASER UNIVERSITY
Fall 2006

APPROVAL

Name: Seyed Mohammad Mahdi Javadi

Degree: Master of Science

Title of thesis: Sparse Modular GCD Algorithm for Polynomials over Algebraic Function Fields

Examining Committee: Dr. Andrei Bulatov
Chair

Dr. Michael Monagan, Senior Supervisor

Dr. Arvind Gupta, Supervisor

Dr. Marni Mishna, Examiner

Date Approved: November 22nd, 2006.

Abstract

Let F = Q(t1,...,tk). For i, 1 <= i <= r, let mi(z1,..,zi) be a monic and irreducible polynomial with coeficients from the field F. Let L = F[z1,..,zr]/ < m1,...,mr>. L is an algebraic function field in k parameters t1,...,tk. Let f1 and f2 be two polynomials in L[x1,..,xn] \ {0}. The problem treated here is the computation of a greatest common divisor of f1 and f2. One way of solving the problem is using the ModGcd algorithm of Monagan and van Hoeij which is an extension of both the modular GCD algorithm of Brown for Z[x1,...,xn] and Encarnacion's algorithm for Q(z)[x] to function fields. ModGcd uses dense interpolation to find the image of the gcd modulo a prime. We introduce the SparseModGcd algorithm which is a modification of ModGcd and takes advantage of Zippel's sparse interpolation algorithm. As a result, SparseModGcd has a better performance when g = gcd(f1,f2) is sparse. SparseModGcd is a Las Vegas algorithm.

Keywords:
Modular Algorithms, Zippel's Sparse Interpolation Algorithm, Polynomial Greatest Common Divisors.

To my dearest parents, Nahid and Ahmad,
and my beloved wife, Maryam

"What we know is a drop, what we don't know, an ocean."

— Isaac Newton

Acknowledgments

I am mostly grateful to my supervisor Dr. Michael Monagan for his generous support and invaluable remarks on my work. I am also deeply indebted to him for everything he has taught me.

I would like to thank my co-supervisor, Dr. Arvind Gupta, for reviewing this thesis.

I would like to give my special thanks to my beloved wife, colleague and best friend, Maryam, whose love, help and tolerance exceeded all reasonable bounds.

Finally, I wish to thank my parents for their unconditional love, continuous supports and many sacrifices throughout my life.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Computing greatest common divisors is an important tool in computer algebra systems (e.g. Maple and Mathematica) with many applications such as simplifying fractions of polynomials and factoring polynomials. Let F be a field and $f_1, f_2 \in F[x_1, ..., x_n]$ be two non-zero polynomials. The problem here is to find the *greatest common divisor* g of f_1 and f_2 which is the polynomial with highest degree that divides both f_1 and f_2.

Example 1.1. Let $f_1 = (x-1)(x+1)(x^3-10)$ and $f_2 = (x-1)(x+1)(x^2+3x-1)$. Here the polynomial x^2-1 divides both of the input polynomials f_1 and f_2. Also the polynomials x^3-10 and x^2+3x-1 are irreducible. Hence we have

$$g = \gcd(f_1, f_2) = x^2 - 1.$$

One of the most important methods for finding the gcd of two univariate polynomials is the *Euclidean algorithm.* It uses the fact that

$$\gcd(f_1, f_2) = \gcd(f_1, r),$$

where r is the remainder of f_2 divided by f_1.

In our previous example, the coefficient field was $\mathbb{Q}$. Our main focus in this thesis is finding the gcd of two polynomials with coefficients in an *algebraic function field.*

Example 1.2. Let $F = \mathbb{Q}(t)$ and $m(z) = z^2 - t$. Let $L = F[z]/\langle m(z)\rangle$. Thus L is an *algebraic function field* of degree 2 in one parameter t where $z = \sqrt{t}$. Let

$$f_1 = x^3 + 2tzx^2 + 5x + zx^2 + 2t^2x + 5z,$$

$$f_2 = x^2 + zxt + zx + t^2$$

be the input polynomials. Here $f_1, f_2 \in L[x]$. To find the gcd of f_1 and f_2 using the Euclidean algorithm, we divide f_1 by f_2 to get the remainder

$$r_1 = \text{rem}(f_1, f_2) = (5 - t^3)x - t^3 z + 5z = (5 - t^3)x + (5 - t^3)z.$$

Since $\gcd(f_1, f_2) = \gcd(f_2, r_1)$, we now proceed with inputs f_2 and r_1, but we first make the remainder r_1 *monic* to reduce expression swell (see Example 1.8). That is, we divide r_1 by its leading coefficient to obtain

$$\text{monic}(r_1) = r_1/(5 - t^3) = x + z.$$

Now we divide f_1 by $x + z$ and the new remainder r_2 is 0. This means that we have found the gcd which is the last remainder, namely

$$\gcd(f_1, f_2) = \text{monic}(r_1) = x + z,$$

and we are done.

Note that the Euclidean algorithm blows up on larger examples and is not directly applicable to multivariate polynomials over a field.

In the next sections of this chapter, we will first give a brief description of the Chinese remainder theorem and polynomial interpolation. Next we describe some known methods of GCD computation for multivariate polynomials with integer coefficients such as the Euclidean pseudo PRS algorithm, Brown's modular GCD algorithm, Zippel's Sparse Interpolation algorithm and the GCDHEU algorithm of Char, Geddes and Gonnet. Finally, we will give a brief description of rational number and function reconstruction and Encarnacion's algorithm for univariate polynomials over a number field.

In Chapter 2 we discuss the problem of polynomial GCD computation over algebraic function fields. We first present Monagan and van Hoeij's *ModGcd* which is a dense algorithm and then we introduce *SparseModGcd*, which is our modification of ModGcd and takes advantage of sparse interpolation for a fast solution in the case where the input polynomials are sparse.

The details of our Maple implementation for our *SparseModGcd* algorithm are described in Chapter 3. We show how different factors such as rational function reconstruction and polynomial evaluation can affect the speed of the algorithm. We also present a running time comparison of the dense and sparse algorithms for different choices of input polynomials.

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

1.1 The Chinese Remainder Theorem

Theorem 1.3. *(The Integer Chinese Remainder Theorem) There exists a unique integer* $0 \leq u < M$ *satisfying*

$$u \equiv u_1 \bmod m_1,$$

$$u \equiv u_2 \bmod m_2,$$

$$\vdots$$

$$u \equiv u_n \bmod m_n,$$

where $m_1, m_2, ..., m_n$ *are pairwise relatively prime integers with* M *as their product and each* $u_i \in \mathbb{Z}_{m_i}$ *is a specified residue.*

Example 1.4. Suppose we want to find the smallest positive integer u such that the set of congruences $\{u \equiv 1 \bmod 2, u \equiv 2 \bmod 3, u \equiv 3 \bmod 5, u \equiv 4 \bmod 7\}$ hold. By solving this system (there are several algorithms, see [6]) we obtain $u = 28$.

If each $m_i < B$, e.g. m_i is a 32 bit prime number, the cost of integer Chinese remaindering with n moduli is $O(n^2)$ using classical algorithms.

For the case where each u_i is a polynomial instead of an integer, one way to determine the solution u from its images is to apply the integer Chinese remainder algorithm (as mentioned above) on each coefficient separately.

Example 1.5. Suppose we want to find the polynomial u with coefficients less than M such that

$$u \equiv 2x^2 + 3x + 1 \bmod 5,$$

$$u \equiv x^2 + 6x + 2 \bmod 7.$$

The answer is

$$u = a_2x^2 + a_1x + a_0,$$

for some integers a_0, a_1 and a_2. To compute a_2 we need to apply the Chinese remainder algorithm on the set of images $\{u_{11} = 2, u_{21} = 1\}$ (the coefficients of x^2 in u_1 and u_2) and the set of moduli $\{m_1 = 5, m_2 = 7\}$. This gives $a_2 = 22$. We do the same thing to find the other two coefficients a_1 and a_0. The final answer is $u = 22x^2 + 13x + 16$.

Note that to recover negative coefficients, we use the symmetric range, i.e. we seek solutions for u satisfying $-\lfloor \frac{M}{2} \rfloor \leq u < \lceil \frac{M}{2} \rceil$. This could be done by first computing the solution in the positive range and then converting it to the symmetric range. Another approach is to do all the operations in the symmetric range during Chinese remaindering algorithm.

1.2 Polynomial Interpolation

Interpolation (in our context) is the process of finding a polynomial, from its images obtained by evaluating at some sample points (see Gathen and Gergard [11]).

Theorem 1.6. *Suppose that $u_1, ..., u_n, v_1, ..., v_n$ in a field F are given. If $u_1, ..., u_n$ are distinct, there exists a unique polynomial $f \in F[x]$ of degree less than n such that $f(u_i) = v_i$ for all i.*

Example 1.7. Suppose that we want to interpolate a polynomial f with $\deg_x(f) = 2$. We need at least three $(\deg_x(f) + 1)$ images to proceed. Given the following images:

$$u_1 = f(x = 1) = 41,\ u_2 = f(x = 2) = 91,\ u_3 = f(x = 3) = 169,$$

the interpolated polynomial, f, is

$$f = 14x^2 + 8x + 19.$$

To determine f, one can solve a system of n linear equations in $O(n^3)$ time, or use *Lagrange Interpolation* or *Newton Interpolation* which both have complexity $O(n^2)$.

1.3 Polynomial GCD Computation

In this section we will introduce some known GCD algorithms with a simple example for each.

1.3.1 The Euclidean Pseudo PRS algorithm

Example 1.8. Suppose we want to find the gcd of the following polynomials

$$f_1 = x^6 + 9x^5 - 3x^4 - 2x^3 + 6x^2 - 3,$$

$$f_2 = 5x^5 + 6x^4 - 6x^3 - 8x^2 + 5.$$

If we try Euclid's algorithm, after the first division we get the first remainder

$$r_1 = \text{rem}(f_1, f_2) = -\frac{279}{25}x^4 + \frac{224}{25}x^3 + \frac{462}{25}x^2 - x - \frac{54}{5}.$$

Now we compute the next remainder

$$r_2 = \text{rem}(f_2, r_1) = \frac{803300}{77841}x^3 + \frac{211075}{25947}x^2 - \frac{446500}{77841}x - \frac{13525}{2883}.$$

Since $r_2 \neq 0$ we continue the algorithm to get the following sequence of remainders

$$r_3 = \text{rem}(r_1, r_2) = \frac{44414751303}{25811635600}x^2 + \frac{12250071693}{3226454450}x - \frac{70406172567}{25811635600},$$

$$r_4 = \text{rem}(r_2, r_3) = \frac{129800856191118800}{2815811338079961}x - \frac{1360431602127677200}{25342302042719649},$$

$$r_5 = \text{rem}(r_3, r_4) = -\frac{208805520823351281995087}{326369520495390530616200},$$

$$r_6 = \text{rem}(r_4, r_5) = 0.$$

Since $r_6 = 0$, a gcd of f_1 and f_2 is r_5. Since a gcd is unique only up to a scalar, $\gcd(f_1, f_2) = 1$.

This example illustrates that Euclid's algorithm is not efficient since we have to deal with fractions and the rapidly growing coefficients after each division. We can improve the efficiency of the algorithm by avoiding the fractions by using *pseudo-division.*

Definition 1.9. Let $f_1, f_2 \in \mathbb{Z}[x]$ with $\deg(f_1) \geq \deg(f_2)$ and $f_2 \neq 0$. Let $\text{lc}(f)$ denote the leading coefficient of f. Let $d = \deg(f_1) - \deg(f_2) + 1$ and $m = \text{lc}(f_2)^d$. The *pseudo-remainder* $\tilde{r}$ and *pseudo-quotient* $\tilde{q}$ of f_1 divided by f_2 are the remainder and quotient, respectively, of mf_1 divided by f_2. They satisfy $mf_1 = \tilde{q}f_2 + \tilde{r}$ and $\tilde{r} = 0$ or $\deg(\tilde{r}) < \deg(f_2)$. Moreover, if $f_1, f_2 \in \mathbb{Z}[x]$ during the division of mf_1 by f_2, no fractions appear, and $\tilde{r}, \tilde{q} \in \mathbb{Z}[x]$ (not $\mathbb{Q}[x]$).

Example 1.10. Let $f_1 = 3x^2 + 2x + 1$ and $f_2 = 2x + 1$. Here $d = 2, m = 2^2 = 4$. Now by dividing mf_1 by f_2, we have $\tilde{q} = 6x + 1$ and $\tilde{r} = 3$.

Let $\text{prem}(a, b)$ denote the pseudo-remainder of a divided by b.

Definition 1.11. (Brown [1]) For non-zero polynomials f_1 and f_2 with $\deg(f_1) \geq \deg(f_2)$, let $f_1, f_2, ..., f_k$ be a sequence of nonzero polynomials such that $f_i = \text{prem}(f_{i-2}, f_{i-1})$ for $i = 3, ..., k$, and $\text{prem}(f_{k-1}, f_k) = 0$. Such a sequence is called a *pseudo polynomial remainder sequence* (pseudo PRS).

In *the Euclidean pseudo PRS algorithm* (see Collins [3]) we use pseudo-division to get a pseudo PRS. Consider the polynomials from the last example. The pseudo PRS in this case is

$$r_1 = \text{prem}(f_1, f_2) = -279x^4 + 224x^3 + 462x^2 - 25x - 270,$$

$$r_2 = \text{prem}(f_2, r_1) = 803300x^3 + 633225x^2 - 446500x - 365175,$$

$$r_3 = \text{prem}(r_1, r_2) = 27759219564375x^2 + 61250358465000x - 44003857854375,$$

$$r_4 = \text{prem}(r_2, r_3) = 27650113246590737208721270312500000x - 321998285740505$$

$$7253647848906250000,$$

$$r_5 = \text{prem}(r_3, r_4) = -78908130690544269677896460007058853018281823654058034147455047607421875000000000,$$

$$r_6 = \text{prem}(r_4, r_5) = 0.$$

This has resolved the problem with fractions, but the coefficient growth is still a serious problem. One can see that the length of the coefficients is doubling at each step!

Let $\text{cont}_x(f)$ be the gcd of the coefficients of f with respect to x and $\text{pp}_x(f) = f/\text{cont}_x(f)$ be the *primitive part* of f with respect to x. One improvement to the Euclidean pseudo PRS algorithm is to make the remainder r_i *primitive* at each step, by setting

$$r_i = \text{pp}(\text{prem}(r_{i-1}, r_{i-2})).$$

This is called the *primitive Euclidean algorithm.* Consider the input polynomials of the last example. If we run the primitive Euclidean algorithm, we will get the following pseudo PRS

$$r_1 = \text{pp}(\text{prem}(f_1, f_2)) = -279x^4 + 224x^3 + 462x^2 - 25x - 270,$$

$$r_2 = \text{pp}(\text{prem}(f_2, r_1)) = 32132x^3 + 25329x^2 - 17860x - 14607,$$

$$r_3 = \text{pp}(\text{prem}(r_1, r_2)) = -570583x^2 + 1258984x - 904487,$$

$$r_4 = \text{pp}(\text{prem}(r_2, r_3)) = 45258957x - 52706137,$$

$$r_5 = \text{pp}(\text{prem}(r_3, r_4)) = -1,$$

$$r_6 = \text{pp}(\text{prem}(r_4, r_5)) = 0.$$

The coefficient growth problem is almost solved for this example. One can show that the growth is linear in the number of steps. But unfortunately this involves many gcd computations (for finding the content of r_i and making it primitive at each step) which consequently slows down the algorithm. When computing a gcd in $\mathbb{Z}[x_1, ..., x_n]$, the recursive gcds to make pseudo-remainders primitive are expensive.

1.3.2 Brown's Modular GCD Algorithm

The best solution to the coefficient growth problem is to use a *modular* algorithm. A modular algorithm projects down the problem to finding the answer modulo a sequence of primes and then builds up the desired answer using the Chinese remainder theorem. *Brown's algorithm* (see [1]) is a modular algorithm for finding the gcd of two multivariate polynomials with coefficients in $\mathbb{Z}$. It also uses polynomial evaluation and interpolation. Since we are computing the gcd modulo a prime p at each step, the coefficients of the polynomials can not be greater than p, therefore the coefficient growth problem will never occur. Consider the following example.

Example 1.12. Suppose we want to find $g = \gcd(f_1, f_2)$ where

$$f_1 = -3x^3y - x^3 - 45xy^2 - 12xy + x,$$

$$f_2 = x^4 + x^2 + 15x^2y + 30y - 2.$$

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

$$f_1(\alpha_1) \bmod p_1 = 7x^3 + 10x,$$

$$f_2(\alpha_1) \bmod p_1 = x^4 + 5x^2 + 6, \text{ and}$$

$$h_1 = \gcd(f_1(\alpha_1), f_2(\alpha_1)) \bmod p_1 = x^2 + 3.$$

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

$$f_1(\alpha_2) \bmod p_1 = 4x^3 + 6x,$$

$$f_2(\alpha_2) \bmod p_1 = x^4 + 9x^2 + 3, \text{ and}$$

$$h_2 = \gcd(f_1(\alpha_2), f_2(\alpha_2)) \bmod p_1 = x^2 + 7.$$

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

$$h = x^2 + 4y + 10.$$

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

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

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

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

Now applying the *Chinese Remainder theorem* to the images g_1 and g_2 we compute a candidate g' for g, the gcd we are seeking. Because in our example g is monic, if this candidate divides both of the input polynomials, then it is equal to g and we are done, otherwise we need to choose another prime p_3 and keep going until we get a candidate which divides both f_1 and f_2. Applying the Chinese remainder theorem results in

$$g' = x^2 + 15y - 1.$$

Since $g'|f_1$ and $g'|f_2$ we conclude that

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

and we are done.

There are many details to this algorithm such as the treatment for unlucky and bad evaluation points, unlucky and bad primes and the leading coefficient construction to make the described idea work in all cases and work efficiently. We postpone this to Chapter 2.

1.3.3 Zippel's sparse interpolation

In Brown's modular GCD algorithm, we first find the images of the gcd modulo a sequence of primes, and then recover the actual gcd from these images using the Chinese remainder theorem. Brown's algorithm will take exponential time in n to interpolate $1+x_1^d+x_2^d+...+x_n^d$, even though this polynomial has only $n+1$ terms. Because sparse polynomials occur quite frequently in practice, several algorithms with time complexity polynomial in d, n and t where t is the number of terms of g have been developed. Zippel in [16] (see also [17] for a more accessible reference) presented a new algorithm, which is basically the same as Brown's algorithm except that after computing the first image $g_1 = g \bmod p_1$ we know the *form* of the actual gcd. That is we know which *terms* are present in g, assuming that g_1 is of the correct form. Now for computing $g_2 = g \bmod p_2$ we only need to find the coefficients for terms in our assumed form of the gcd corresponding to g_2. Consider the following example.

Example 1.13. Suppose the two input polynomials are

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

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

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

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

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

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

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

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

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

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

From this we get the following linear equations modulo 13

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

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

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

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

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

So we have

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

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

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

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

1.3.4 The GCDHEU Algorithm

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

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

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

Example 1.14. Let

$$f_1 = 6x^4 + 21x^3 + 38x^2 + 33x + 14,$$

$$f_2 = 12x^4 - 3x^3 - 14x^2 - 39x + 28.$$

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

$$f_1(\xi = 1000) = 6021038033014,$$

$$f_2(\xi = 1000) = 11996985961028.$$

Notice how the coefficients of f_1 and f_2 appear in the evaluations. Next we calculate the integer gcd of $f_1(\xi = 1000)$ and $f_1(\xi = 1000)$ (using the Euclidean algorithm) to get

$$\gcd(6021038033014, 11996985961028) = 6012014.$$

Notice that this corresponds to the polynomial $h = 6x^2 + 12x + 14$ with $x = 1000$. If we divide h by its content (the gcd of its coefficients) we get

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

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

There are some more GCD computation methods such as the *EEZ-GCD* which is developed by Wang (see [13]) and is a modular algorithm. Another method which is presented by Encarnacion is called *Encarnacion's algorithm* (see [5]) which is used to compute the gcd of two polynomials over one algebraic number field. A brief description of this algorithm is given in Section 1.5

1.4 Rational Number and Function Reconstruction

Definition 1.15. Let $n/d \in \mathbb{Q}$ with $\gcd(n, d) = 1$, and let m be a positive integer satisfying $\gcd(m, d) = 1$. Let $u \equiv n/d \bmod m$. The rational reconstruction problem is given u and m find n and d.

Recall that on input of $m > u \geq 0$, the Euclidean algorithm computes a sequence of triples $s_i, t_i, r_i \in \mathbb{Z}$ satisfying

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

Hence we have

$$t_i u \equiv r_i \pmod{m}.$$

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

Example 1.16. For $m = 13$ and $u = 8$ we have

$$S = \{\frac{8}{1}, \frac{5}{-1}, \frac{3}{2}, \frac{2}{-3}, \frac{1}{5}\} = \{8, -5, \frac{3}{2}, \frac{2}{-3}, \frac{1}{5}\}.$$

Wang *et al.* (see [14] and [15]) show that if $m > 2|nd|$ and $\gcd(m, d) = 1$ then n/d appears in S.

One can use either Wang's algorithm [14] or Monagan's algorithm [8] to select the rational from the set S. Both of these algorithms have time complexity $O(\log^2 m)$. Wang's algorithm succeeds and outputs n/d when $m > 2(\max(|n|, |d|))^2$. Monagan's algorithm succeeds when m is a few bits longer than $2|n|d$ with high probability.

Definition 1.17. Let F be a field and let $m, u \in F[x]$ where $0 \leq \deg(u) < \deg(m)$. The problem of *Rational Function Reconstruction* is given m and u, find a rational function $n/d \in F(x)$ such that

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

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

Given polynomials m and u as specified in the above definition, again the Extended Euclidean algorithm finds all solutions satisfying $\deg(n)+\deg(d) < \deg(m)$ up to multiplication by scalars.

Example 1.18. Let $F = \mathbb{Z}_7, u = x^2 + 5x + 6$ and $m = (x - 1)(x - 2)(x - 3)$. Using the Extended Euclidean Algorithm we get the following set of solutions:

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

The solution to the rational function reconstruction is not always unique. We can force the uniqueness by choosing degree bounds $\deg(n) \leq N$ and $\deg(d) \leq D$ satisfying $N + D < \deg(m)$. As an example, if we choose degree bounds $N = 1$ and $D = 1$ in Example 1.18, the unique answer is

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

1.5 Encarnacion Algorithm

Suppose $f_1, f_2 \in \mathbb{Z}[x]$ and $g = \gcd(f_1, f_2) = 3x^2 + 2x - 1$. When we compute $\gcd(f_1, f_2)$ mod $p_1 = 11$, the gcd is unique up to a scalar multiple in $\mathbb{Z}_p$. We usually take the monic

gcd. For $p_1 = 11$ and $p_2 = 13$ we obtain

$$\gcd(f_1, f_2) \bmod 11 = x^2 + 8x + 7, \gcd(f_1, f_2) \bmod 13 = x^2 + 5x + 4.$$

When we apply the Chinese remainder theorem, we obtain a monic image $x^2 + 96x + 95$. This is the image of $g/\mathrm{lc}(g) = x^2 + \frac{2}{3}x - \frac{1}{3}$. Thus to recover g, we first recover the rational coefficients of $g/\mathrm{lc}(g)$, namely $\{1\ ,\ \frac{2}{3}, -\frac{1}{3}\}$ from $\{1, 96, 95\}$ mod 11×13. We use rational number reconstruction to do this. Then we clear denominators.

Encarnacion [5] used rational number reconstruction to compute the gcd of $f_1, f_2 \in L[x]$ where L is a number field.

Example 1.19. Let $z = \sqrt{2}$ and $L = \mathbb{Q}(z)$. Let the input polynomials be

$$f_1 = 3\,x^4 + (2\,z - 2)\,x^3 + (-1 + 4\,z)\,x^2 + (4 - 2\,z)\,x - z + 2,$$

$$f_2 = 6\,x^3 + (-1 + 4\,z)\,x^2 + (-4 + 4\,z)\,x - 1 + z.$$

In this example $g = 3\,x^2 + (2\,z - 2)\,x + z - 1$ is the gcd of f_1 and f_2 in $L[x]$. To compute g, we first compute the image of g modulo $p = 11$ to obtain

$$h = x^2 + (8\,z + 3)\,x + 4\,z + 7.$$

If we apply the rational number reconstruction on the coefficients of h we get

$$h' = x^2 + \left(\frac{2}{3}\,z - \frac{2}{3}\right)x + \frac{1}{3}\,z - \frac{1}{3}.$$

After clearing the denominators we obtain

$$g_{11} = 3h' = 3\,x^2 + (2\,z - 2)\,x + z - 1.$$

A technical difficulty is that some primes may result in *zero divisors*. For example, let $f_1, f_2 \in \mathbb{Q}(z)[x]/\langle m(z)\rangle$ and $f_1 = x^4 + (z-2)x^2 + zx + 1$, $f_2 = (z-3)x^3 + x + 2z$ and $m(z) = z^2 - 2$. If we choose $p_1 = 7$ and we divide f_1 mod p_1 by f_2 mod p_1, we hit a zero divisor while we are trying to invert $\mathrm{lc}(f_2) = z - 3 \mod m(z) = (z-3)(z+3)$. The solution to this problem is to use another prime. Note that there are only finitely many primes p that can cause this problem because these primes must divide $R = \mathrm{res}_z(m(z), m'(z)) \in \mathbb{Z}$, where *res* denotes the resultant.

Chapter 2

GCD Computation over Algebraic Function Fields

In this chapter we consider the problem of computing a gcd of two polynomials over an algebraic function field L. A modular algorithm for computing a gcd for the case of one field extension was developed by Monagan and van Hoeij in [10]. Their algorithm uses dense interpolation. We introduce a modular algorithm which uses sparse interpolation and consequently is much better on sparse polynomials and remains competitive on dense polynomials.

In Section 2.1 we describe the problem. In Section 2.2 we present the dense algorithm of Monagan and van Hoeij. Our sparse algorithm will be described in Section 2.3.

2.1 Definition of the Problem

Let $F = \mathbb{Q}(t_1, ..., t_k)$. For i, $1 \leq i \leq r$, let $m_i(z_1, .., z_i) \in F[z_1, ..., z_i]$ be monic and irreducible over $F[z_1, ..., z_{i-1}]/\langle m_1, ..., m_{i-1}\rangle$. Let $L = F[z_1, .., z_r]/\langle m_1, ..., m_r\rangle$. L is an algebraic function field in k parameters $t_1, ..., t_k$. Suppose that f_1 and f_2 are non-zero polynomials in $L[x_1, .., x_n]$. Let g be the monic gcd of f_1 and f_2. Our problem is, given f_1 and f_2 to compute g or an associate (scalar multiple) of g.

The followings are some definitions from [10]:

Definition 2.1. Let $D = \mathbb{Z}[t_1, ..., t_k]$. A non-zero polynomial in $D[z_1, .., z_r, x_1, .., x_n]$ is said to be *primitive* with respect to $(z_1, .., z_r, x_1, .., x_n)$ if the gcd of its coefficients in D is 1.

Let f be non-zero in $L[x_1, .., x_n]$. The *denominator* of f is the polynomial $\text{den}(f) \in D$ of least total degree in $(t_1, ..., t_k)$ and with smallest integer content such that $\text{den}(f)f$ is in $D[z_1, .., z_r, x_1, .., x_n]$.

The *primitive associate* $\check{f}$ of f is the associate of $\text{den}(f)f$ which is primitive in $D[z_1, .., z_r, x_1, .., x_n]$ and has positive leading coefficient in a term ordering.

Example 2.2. Let $f = 3tx^2 + 6tx/(t^2-1) + 30tz/(1-t)$ where $m_1(z) = z^2 - t$. We have $\text{den}(f) = t^2-1$ and $\check{f} = (t^2-1)x+2x-10z(t+1)$. Here $f \in L[x]$ where $L = \mathbb{Q}(t)[z]/\langle z^2 - t\rangle$ is an algebraic function field in one parameter t.

2.2 Dense Algorithm

This algorithm which is developed by Monagan and van Hoeij is described in [10]. Their algorithm assumes that there is only one minimum polynomial $m(z) \in F[z]$ and one variable x. Later in Section 2.2.7 we will show how to deal with multivariate polynomials. Also our examples use s and t for parameters and not $t_1, t_2, ...$ Their algorithm computes the primitive associate $\check{g}$. Here is an example from Monagan in [10]:

Example 2.3. Let $z = \sqrt{t}$ i.e. $m(z) = z^2 - t$. Suppose that the input polynomials are

$$f_1 = x^2 + \frac{-2t+3}{3}zx + \frac{5}{t}x + \frac{5}{t}z + \frac{-2t^2}{3},$$

$$f_2 = zx^2 + \frac{5}{t}zx + \frac{3-2t^2}{3}x + \frac{-2t}{3}z + \frac{5}{t}.$$

The algorithm first computes

$$\check{f}_1 = 3tx^2 + (-2t^2+3t)zx + 15x + 15z - 2t^3,$$

$$\check{f}_2 = 3tzx^2 + 15zx + (-2t^3+3t)x - 2t^2z + 15.$$

It then computes the $\gcd(\check{f}_1, \check{f}_2)$ modulo a sequence of primes. Let's start with the prime $p = 11$ (on a 64 bit machine, their implementation uses 31 bit primes, but for this example we choose small primes). We obtain

$$\check{g}_{11} = \gcd(\check{f}_1, \check{f}_2) \bmod 11 = tx + 3t^2z + 5.$$

If we apply the rational number reconstruction to the coefficients of g_{11} modulo 11, it fails. So we choose a new prime $q = 13$. We obtain

$$\check{g}_{13} = \gcd(\check{f}_1, \check{f}_2) \bmod 13 = tx - 5t^2z + 5.$$

By applying the Chinese remainder theorem we obtain

$$g_m = tx + 47t^2z + 5 \bmod 11 \times 13.$$

Now we apply the rational reconstruction to the coefficients of g_m modulo $m = 143$. This time it succeeds. The output is

$$h = tx - \frac{2}{3}t^2z + 5.$$

Clearing the denominator results in

$$\check{h} := 3h = 3tx - 2t^2z + 15.$$

Since $\check{h}|\check{f}_1$ and $\check{h}|\check{f}_2$ then $\check{h} = \check{g}$ and we are done. Now we have to show how we compute g_{11} and g_{13}. This is done by computing $\check{g}_{11}$ and $\check{g}_{13}$ at a sequence of evaluation points for t in $\mathbb{Z}_{11}$ and $\mathbb{Z}_{13}$ respectively and applying polynomial interpolation then rational function reconstruction to get the final result.

Suppose we start with $t = 2$ (The algorithm uses random numbers from $[0, p)$ but for this example we will use $t = 2, 3, ...$). We run the Euclidean algorithm modulo $p = 11$ to get

$$g_1 = \gcd(f_1(2, x), f_2(2, x)) \bmod 11 = x - 5z - 3.$$

Now we apply the rational function reconstruction to the coefficients of g_1 in x and z which succeeds with output $h = g_1$, but the output does not divide $\check{f}_1$ modulo 11 so we need more evaluation points. Using $t = 3$ we obtain

$$g_2 = \gcd(f_1(3, x), f_2(3, x)) \bmod 11 = x - 2z - 2.$$

Applying the polynomial interpolation to get the gcd modulo $(t - 2)(t - 3)$ results in

$$c = x + (3t)z + (t - 5).$$

Again, the output of rational function reconstruction applied to c does not divide $\check{f}_1$ and $\check{f}_2$ modulo 11, so we need another evaluation point. Using $t = 4$ we obtain

$$g_3 = \gcd(f_1(4, x), f_2(4, x)) \bmod 11 = x + z + 4.$$

After interpolating the new point we obtain

$$c = x + 3tz - 3t^2 + 5t - 1,$$

which is the gcd of $\check{f}_1$ and $\check{f}_2$ modulo $(t-2)(t-3)(t-4)$. Now we apply rational function reconstruction to the coefficients of c to obtain linear numerators and linear denominators

$$h = x + \frac{3t}{1}z + \frac{5}{t}.$$

Clearing the denominators in t we obtain

$$\check{h} = th = tx + 3t^2z + 5.$$

Since $\check{h}|\check{f}_1(t,x) \bmod 11$ and $\check{h}|\check{f}_2(t,x) \bmod 11$ then $\check{h} = \check{g}_{11}$ and we are done.

2.2.1 Unlucky and Bad Primes, Unlucky and Bad Evaluation Points

Recall that $f_1, f_2 \in L[x]$ and g is their monic gcd. As we saw in Example 2.3 this modular GCD algorithm computes a gcd of $\check{f}_1$ and $\check{f}_2$ by computing the $\gcd(\check{f}_1, \check{f}_2)$ modulo a sequence of primes and modulo a sequence of evaluation points. The algorithm then reconstructs $\check{g}$ from these images. Only images which are computed modulo *good* primes and *good* evaluation points can be used during the reconstruction for it to be successful. However not all primes and evaluation points are good.

Definition 2.4. A prime p is a *good* prime if $\check{g}_p = \gcd(\check{f}_1 \bmod p, \check{f}_2 \bmod p)$ exists and $\text{monic}(\check{g}_p) = \text{monic}(\check{g} \bmod p)$. Similarly an evaluation point $\alpha \in \mathbb{Z}^k$ is a *good* evaluation point if $\check{g}_\alpha = \gcd(\check{f}_1(\alpha), \check{f}_2(\alpha))$ exists and $\text{monic}(\check{g_\alpha}) = \text{monic}(\check{g}(\alpha))$.

Definition 2.5. Suppose $f_1, f_2 \in L[x]$. A prime p is said to be a *bad prime* if the leading coefficient of $\check{f}_1$ or $\check{f}_2$ with respect to x or any $\check{m}_i$ with respect to z_i vanishes mod p.

Example 2.6. Suppose that

$$\check{f}_1 = 28tx^3 + 19ztx + 2t^2 + 10 \text{ and } \check{f}_2 = 52zx^2 + 10x + zt^3 - t.$$

All $p_1 = 2$ and $p_2 = 7$ and $p_3 = 13$ are bad primes.

Example 2.7. Suppose that

$$\check{f}_1 = (x + t^2z^3 - t^2 + 1)(2x^3 + 1) \text{ and } \check{f}_2 = (x + t^2z^3 - t^2 + 1)(x^3 + 1)$$

and $\check{m}(z) = 7z^5 + 1$. Here $\check{g} = \gcd(\check{f}_1, \check{f}_2) = x + t^2z^3 - t^2 + 1$. Modulo $p = 7$, $\check{m}(z)$ becomes 1. Hence the image of the gcd modulo 7 would be 0 which is not good for reconstructing $\check{g}$.

The good thing about *bad primes* is that they can be ruled out in advance.

Definition 2.8. Suppose $f_1, f_2 \in L[x]$. A prime p is said to be *unlucky* if $\check{g}_p = \gcd(\check{f}_1, \check{f}_2)$ mod p has a higher degree in x than the actual gcd $\check{g}$.

Example 2.9. Consider the input polynomials

$$\check{f}_1 = (x+z)(x+17t+t^2+z) \text{ and } \check{f}_2 = (x+t)(x+t^2+z).$$

Here we have $\check{g} = \gcd(\check{f}_1, \check{f}_2) = 1$ but $\check{g}_{17} = \gcd(\check{f}_1, \check{f}_2)$ *mod* $17 = x + t^2 + z$ which obviously has a higher degree than $\check{g}$, so $p = 17$ is an unlucky prime.

The same problems that can happen for primes also happen for evaluation points.

Definition 2.10. Suppose $f_1, f_2 \in L[x]$. An evaluation point $t_1 = \alpha$ is called a *bad evaluation point* if the degree of $\check{f}_1$ or $\check{f}_2$ with respect to x or any $\check{m}_i$ with respect to z_i decreases after evaluating at this point.

Example 2.11. Let $\check{f}_1(t, x) = 3(4t-1)x^3 + zt^3x + 10t$ and $p = 17$. Here $t = 13$ is a bad evaluation point because $\check{f}_1(13, x)$ mod $p = zt^3x + 10t$ has lower degree than $\check{f}_1$.

Definition 2.12. Suppose $f_1, f_2 \in L[x]$. An evaluation point $t_1 = \alpha$ is said to be *unlucky* if $\gcd(\check{f}_1(\alpha), \check{f}_2(\alpha))$ mod p has a higher degree in x than the actual gcd g.

Example 2.13. Let $\check{f}_1 = x^2 + (t-1)x + 18z$ and $\check{f}_2 = x^2 + 18z$. When computing the $\gcd(\check{f}_1, \check{f}_2)$ mod 11, if we choose the evaluation point $t = 1$, we get $\gcd(\check{f}_1(1), \check{f}_2(1))$ mod $11 = x^2 + 7z$ but $\gcd(\check{f}_1, \check{f}_2)$ mod $11 = 1$. Hence the evaluation point $t = 1$ is unlucky.

Like bad primes, bad evaluation points can be determined and discarded beforehand, but unlucky ones can not.

2.2.2 Zero Divisors

Recall that a non-zero element α of a ring R is a zero divisor if there exist a non-zero element $\beta \in R$ s.t. $\alpha\beta = 0$.

When we are trying to compute the gcd of $f_1(\alpha_1, ..., \alpha_k, x), f_2(\alpha_1, ..., \alpha_k, x)$ $(\alpha_1, ..., \alpha_k$ are the evaluation points) with the Euclidean algorithm we might encounter a zero divisor, in which case the Euclidean algorithm fails (see [9]). The bigger the prime p is, the smaller the chance of hitting a zero divisor would be.

Example 2.14. Let $f_1 = (z+2t)x^2 + tx + z$, $f_2 = zx^3 + tx^2 + (z-2)x + 8t$ be the input polynomials and $m(z) = z^2 - t$. Suppose we choose the first prime $p = 7$. If we evaluate the inputs at $t = 2$ we obtain $f_1' = (z-3)x^2 + 2x + z$ and $f_2' = zx^3 + 2x^2 + (z-2)x + 2$. When we run the Euclidean algorithm on the inputs f_1' and f_2' we hit a zero divisor while trying to invert $\mathrm{lc}(f_1') = z - 3$. Note $z^2 - 2 = (z-3)(z+3) \bmod 7$.

Unlucky primes must be avoided if g is to be correctly reconstructed. Unlike bad primes, unlucky primes can not be detected and discarded in advance. Brown in [1] showed how to do this in a way that is efficient for $\mathbb{Z}[x]$. Whenever an image of the gcd does not have the same degree, we keep only those images of smallest degree and discard the others. His strategy is based on the following lemma.

Lemma 2.15. (see Geddes *et al.* [6]) Let R and S be two unique factorization domains and $A, B \in R[x]\backslash\{0\}$ and $G = \mathrm{GCD}(A, B)$. Let $\phi : R \to S$ be a ring morphism and $\phi : R[x] \to S[x]$ be the natural extension to $R[x]$ and $H = \mathrm{GCD}(\phi(A), \phi(B))$. If $\phi(\mathrm{lc}(A)) \neq 0$ then $\deg(H) \geq \deg(G)$. Moreover, if $\phi(\mathrm{lc}(A)) \neq 0$ and $\deg(H) = \deg(G)$ then $\phi(G) = uH$ for some scalar $u \in S$.

Proof: $H = \mathrm{GCD}(\phi(A), \phi(B)) = \mathrm{GCD}(\phi(\bar{A}G), \phi(\bar{B}G))$ for some $\bar{A}, \bar{B} \in R[x]$. Hence $H = \phi(G)\mathrm{GCD}(\phi(\bar{A}), \phi(\bar{B})) \Rightarrow \phi(G)|H$ (provided $\phi(G) \neq 0$).

Now we want to prove that $H \neq 0$. Since R is a unique factorization domain $\mathrm{lc}(A) = \mathrm{lc}(\bar{A}G) = \mathrm{lc}(\bar{A})\mathrm{lc}(G)$. Since $\phi(\mathrm{lc}(A)) \neq 0$ we have $\phi(\mathrm{lc}(\bar{A})\mathrm{lc}(G)) \neq 0 \Rightarrow \phi(\mathrm{lc}(\bar{A}))\phi(\mathrm{lc}(G)) \neq 0$. Since S is a unique factorization domain, $\phi(\mathrm{lc}(\bar{A})) \neq 0$ and $\phi(\mathrm{lc}(G)) \neq 0$ and we conclude that $\mathrm{GCD}(\phi(\bar{A}), \phi(\bar{B})) \neq 0$ and $\phi(G) \neq 0$, hence $H \neq 0$.

S is an integral domain and we have proved that $\phi(G)|H$ and $H \neq 0$, thus we have $\deg(H) \geq \deg(G)$. Also if $\deg(H) = \deg(G)$ then $u = \mathrm{GCD}(\phi(\bar{A}), \phi(\bar{B}))$ must be a constant, so we have $\phi(G) = uH$ and $u \in S$. This completes the proof.

For $R = \mathbb{Z}[x]$ Brown uses $\phi_p(f) = f \bmod p$ and the lemma holds. In our case $R = L[x]$ and $\phi_p : L \to L \bmod p$ so $S = L[x] \bmod p$ is usually not a unique factorization domain. Monagan and van Hoeij in [10] have generalized the lemma as follows.

Theorem 2.16. *Let* $f_1, f_2 \in L[x]$ *be two non-zero polynomials where* $L = F[z_1, .., z_r]/\langle m_1, ..., m_r \rangle$ *and* $F = \mathbb{Q}(t_1, ..., t_k)$. *Let* $\check{g} = \gcd(\check{f}_1, \check{f}_2)$. *Let* p *be a prime and* $\alpha = (t_1 = \alpha_1, ..., t_k = \alpha_k)$. *Suppose that the Euclidean algorithm applied to* $\check{f}_1(\alpha, x)$ *and* $\check{f}_2(\alpha, x)$ *modulo* p *does not fail and outputs* g_p. *If* α *is not a bad evaluation point and* p *is*

not a bad prime, $\deg_x(g_p) \geq \deg_x(\check{g})$. *Moreover if* $\deg_x(g_p) = \deg_x(\check{g})$ *then* $\text{monic}(g_p) = \text{monic}(\gcd(\check{f}_1(\alpha, x), \check{f}_2(\alpha, x)) \bmod p)$.

Proof: See Monagan and van Hoeij [10]. The only difference is the number of field extensions but this has no significant change in the proof in [10].

2.2.3 Termination Conditions

When we are dealing with a modular algorithm, we always encounter the problem of when to stop the algorithm, i.e., when do we have enough images to construct the actual gcd from its images.

One approach toward solving this problem is to determine an upper bound for the number of images we need before starting the algorithm. Unfortunately, we can not compute a good upper bound efficiently based on the size of the inputs and we may end up wasting time, computing a lot of extra images. This is because $\check{g}$ can be very small compared to $\check{f}_1$ and $\check{f}_2$.

The modular algorithm ModGcd, which is described in this section stops when the reconstructed result $\check{h}$ does not change from one prime (evaluation point) to the next and then tests if $\check{h}|\check{f}_1$ and $\check{h}|\check{f}_2$. Then Theorem 2.16 implies $\check{h} = g$. This means that ModGcd algorithm is *output sensitive*, i.e., the number of primes (evaluation points) used depends on the size of $\check{g}$ and not on any bounds based on the sizes of $\check{f}_1$ and $\check{f}_2$ with high probability.

2.2.4 Algorithm ModGcd

We now present the ModGcd algorithm which is developed by Monagan and van Hoeij in [10]. This modular GCD algorithm first calls subroutine M which computes the GCD in $L[x]$ from a number of images in $L_p[x]$. Subroutine P which is called by subroutine M computes the GCD in $L_p[x]$ from a number of images in $\mathbb{Z}_p(t_1, ..., t_{k-1})[z_1, .., z_r, x]/\langle m_1, ..., m_r \rangle$. In Section 2.2.7 we will show how we can extend the algorithm for multivariate polynomials. Except for the treatment of zero divisors, the algorithm follows Example 2.3.

Remark 2.17. The names which are used for the subroutines M and P in this algorithm are based on the names Brown has used in his modular algorithm. See [1].

Algorithm ModGcd

Input: $f_1, f_2 \in L[x]$ and $m_1, ..., m_r (m_i \in F[z_1, ..., z_i]$ for $1 \leq i \leq r)$.
Output: $\check{g}$, where g is the monic gcd of f_1 and f_2 in $L[x]$.

1. Call Subroutine M with input $\check{f}_1$, $\check{f}_2$ and $\check{m}_1$,...,$\check{m}_r$.

Subroutine M

Input: $f_1, f_2 \in D[z_1, ...z_r]/\langle m_1, .., m_r \rangle [x]$ and $m_1, ..., m_r \in D[z_1, ...z_r]$, $D = \mathbb{Z}[t_1, ..., t_k]$.
Output: $\check{g}$, where g is the monic gcd of f_1 and f_2 in $L[x]$.

1. Set $n = 1$, $G = 0$.
2. Main Loop: Take a new prime p_n
3. Check if p_n is a bad prime, if it is go back to step 2.
4. Let $g_n \in D_{p_n}[z_1, .., z_r, x]$ be the output of subroutine P applied to $f_1, f_2, m_1 \bmod p, ...,$ $m_r \bmod p$.
5. If $g_n =$ "failed" then go back to step 2.
6. If $g_n = 1$ then return 1.
7. If $G = 0$ then set $G = g_n$ and $m_c = p$ then go to step 11.
8. If $\deg_x(g_n) < \deg_x(G)$ then set $G = g_n$,$m_c = p$ then go to step 11. /*All previous primes where unlucky */
9. If $\deg_x(g_n) > \deg_x(G)$ then go back to step 2. /* p_n is an unlucky prime */
10. Select from $\{g_1, ..., g_n\}$ those with the same leading term (in pure lexicographic order with $x > t_1 > ... > t_k$) as g_n. Combine them using Chinese remaindering to obtain G mod m_c.
11. Set $n = n + 1$.
12. Apply integer rational reconstruction to obtain h from G mod m_c. If this fails, go back to step 2.
13. Clear fractions in $\mathbb{Q}$: Set $h = \check{h}$.
14. Trial division: If $h|f_1$ and $h|f_2$ then return h, otherwise, go back to step 2.

Subroutine P

Input: $f_1, f_2 \in D_p[z_1, ...z_r]/\langle m_1, .., m_r\rangle [x]$ and $m_1, ..., m_r \in D_p[z_1, ...z_r]$.
Output: Either $\check{g}$ or "failed" if the algorithm fails to compute the primitive associate of the monic gcd of f_1 and f_2.

0. If k (The number of parameters) $= 0$ then output the result of the Euclidean algorithm applied to f_1, f_2. /* If it fails, then output "failed". */
1. Set $n = 1$, $d = 1$, $G = 0$.
2. Main Loop: Take a new evaluation point α_n.
3. Check if α_n is a bad evaluation point, if it is go back to step 2.
4. Let $g_n \in \mathbb{Z}_p[t_1, .., t_{k-1}][z_1, .., z_r, x]$ be the output of subroutine P applied to $f_1, f_2, m_1, ...,$ m_r at $t_k = \alpha_n$.
5. If $g_n =$ "failed" then
   5.1. Set $d = d + 1$.
   5.2. If $d > n$ output "failed", else go back to step 2.
6. If $g_n = 1$ then return 1.
7. If $G = 0$ then set $G = g_n$ and $m_c = t_k - \alpha_n$ then go to step 11.
8. If $\deg_x(g_n) < \deg_x(G)$ then set $G = g_n$, $m_c = t_k - \alpha_n$ then go to step 11. /*All previous evaluation points where unlucky */
9. If $\deg_x(g_n) > \deg_x(G)$ then go back to step 2. /* α_n is an unlucky evaluation point */
10. Select from $\{g_1, ..., g_n\}$ those with the same leading term in $x, t_1, ..., t_{k-1}$ as g_n. Chinese remainder those to obtain G mod $m_c(t_k)$.
11. Set $n = n + 1$.
12. Apply rational function reconstruction to the coefficients of G in t_k to obtain $h \in \mathbb{Z}_p(t_k)[t_1, ..., t_{k-1}][z_1, ..., z_r, x]$ s.t. $h \equiv G$ mod $m_c(t_k)$. If this fails go back to step 2.
13. Clear fractions in $\mathbb{Z}_p(t_k)$: Set $h = \check{h}$.
14. Trial division: if $h|f_1$ and $h|f_2$ then return h, otherwise, go back to step 2.

2.2.5 Treatment of Zero Divisors

Consider the following example from [10]: Suppose $m(z) = z^2 + 7t - 1$ and $f_1 = x^2 + t$ and $f_2 = (z + 1)x + t$. If subroutine M chooses the prime $p = 7$, we will have $m_p(z) = z^2 - 1$. Since $\mathrm{lc}_x(f_2) = z + 1 | m_p(z)$, the Euclidean algorithm will always fail, while trying to invert $\mathrm{lc}_x(f_2)$ which is a zero divisor for any choice of $t = \alpha$, $\alpha \in \mathbb{Z}_7$. For any other prime $p \neq 7$,

the Euclidean algorithm hits a zero divisor only for the evaluation point $\alpha = 0$. This means that, if we choose the evaluation points at *random* the probability of hitting a zero divisor would be $1/p$.

Monagan and van Hoeij solve the problem of hitting a zero divisor repeatedly using the following strategy. The variable d, in subroutine P, counts the number of times which the Euclidean algorithm fails, which is the number of times the algorithm encounters a zero divisor. The case where $d > n$ happens when the algorithm encounters a lot of zero divisors. This could relate to our choice of prime number or a previous evaluation point.

Note that if most evaluation points are good, and if subroutine P has already computed many good images, then the test $d > n$ prevents, with high probability, that few unlucky choices in step 2 could cause a lot of useful work to be lost.

2.2.6 Trial Divisions

In Step 14 of subroutines M and P, the algorithm uses *trial division* to test whether it has computed the correct gcd. The only difference is that in subroutine P, the trial divisions take place in characteristic p. In [9] Monagan, van Hoeij presented an algorithm for doing trial divisions (in characteristic p) of polynomials in $\mathbb{Z}[z][x]$ modulo $m(z) \in \mathbb{Z}[z]$ which uses pseudo-division and some gcds in $\mathbb{Z}$ to minimize growth of the integer coefficients. We essentially use the same method for our algorithm in Section 2.3, except that the coefficient ring is $D_p = \mathbb{Z}_p[t_1, ..., t_k]$ instead of $\mathbb{Z}$. The same algorithm can also be used for subroutine M with D_p replaced by D. Here we show how to extend it to treat multiple field extensions.

Algorithm Trial Division with Multiple field extensions.

Input: $A, B \in D_p[z_1, ...z_r]/\langle m_1, .., m_r\rangle\,[x]$ and $m_1, ..., m_r \in D_p[z_1, ...z_r]$, $B \neq 0$.
Output: True if $B|A$, False otherwise.

1. Set $m = \deg_x(A), n = \deg_x(B)$.
2. Set $d_1 = \deg_{z_1}(m_1), ..., d_r = \deg_{z_r}(m_r)$.
3. Set $l_b = \mathrm{lc}_x(B)$.
4. Set $l_{m_1} = \mathrm{lc}_{z_1}(m_1), ..., l_{m_r} = \mathrm{lc}_{z_r}(m_r)$.
5. Set $R = A$.
6. While $r \neq 0$ and $m \geq n$ do
   6.1. Set $l_R = \mathrm{lc}_x(R)$.

6.2. Set $g = \gcd(\text{cont}_{z_1,\ldots,z_r}(l_R), l_b)$ *mod* p.

6.3. Set $l_R = l_R/g, s = l_b/g$.

6.4. Set $t = l_R x^{m-n}$.

6.5. Set $R = sR - tb$.

6.6. for i from 1 to r do

6.6.1. while $R \neq 0$ and $\deg_{z_i}(R) \geq d_i$ do

6.6.1.1. Set $l_R = \text{lc}_{z_i}(R)$.

6.6.1.2. Set $g = \gcd(\text{cont}_x(l_R), l_{m_i})$ *mod* p.

6.6.1.3. Set $l_R = l_R/g$.

6.6.1.4. Set $t = l_R z_i^{\deg_{z_i}(R)-d_i}$.

6.6.1.5. Set $R = (l_{m_i}/g)R - tm_i$.

6.7. Set $m = \deg_x(R)$.

7. If $R \neq 0$ then return False, otherwise, return True.

Note that $\deg_{z_j}(m_i) = 0$ if $j > i$. The outer loop reduces the degree of the remainder R in x. In the inner loops, for each i, the algorithm reduces the degree of R in z_i to be less than the degree of m_i in z_i.

2.2.7 Multivariate Polynomials and Non-trivial Content

In the previous examples for the ModGcd algorithm, we assumed that f_1 and $f_2 \in L[x]$, i.e. there is only one variable, x. In [10] Monagan and van Hoeij proposed a simple method for dealing with multivariate input polynomials. Let $f_1, f_2 \in L[x_1, ..., x_n]$ with $n > 1$. In order to compute the gcd of f_1 and f_2 we may consider $f_1, f_2 \in K[x_n]$, where $K = L[x_1, ..., x_{n-1}]$. So we treat the inputs as polynomials in x_n with coefficients in K. Now recall that $\gcd(f_1, f_2) = cb$ where $c = \gcd(\text{cont}_{x_1,\ldots,x_{n-1}}(f_1), \text{cont}_{x_1,\ldots,x_{n-1}}(f_2))$ and $b = \gcd(f_1/c, f_2/c)$. As you see, computing c, requires calling the ModGcd algorithm recursively with one less variable. To compute b, we simply treat $x_1, ..., x_{n-1}$ as parameters!, i.e. we write the input polynomials in $K[x_n]$ where $K = G[z_1, .., z_r]/\langle m_1, ..., m_r \rangle$ and $G = \mathbb{Q}(t_1, ..., t_k, x_1, ..., x_{n-1})$. Now we compute $\check{g}$ using ModGcd. However $\check{g}$ could have a non-trivial content in x_n which needs to be computed and divided out to get b. Here is an example from [10] illustrating this.

Example 2.18. Suppose we have computed $\check{g} = (x_1^2 - s)x_2 - z + x_1$ where $z = \sqrt{s}$. Here $z - x_1$ is the non-trivial content of $\check{g}$. We need to divide $\check{g}$ by this content to get $b = (x_1 + z)x_2 + 1$.

2.3 Sparse Algorithm

In the previous section, we described the ModGcd algorithm which is a *dense* modular algorithm for finding the GCD of two polynomials over algebraic function fields. Suppose $d_i = \deg_{t_i}(g)$. Then ModGcd calls the Euclidean algorithm (in the first step of subroutine P) approximately $O(\prod_{i=1..k} d_i)$ times. Now suppose we have two input polynomials f_1 and f_2 such that $\breve{g} = x_1^2 + t_1^{100}zx + t_2^{100}x + t_3^{100}$. Since $d_1 = d_2 = d_3 = 100$, ModGcd calls the Euclidean algorithm approximately 1000000 times. This motivates us to use *sparse interpolation* instead of *dense interpolation* in subroutine P, because the number of times we need to call Euclidean algorithm in sparse interpolation, depends on the number of terms present in the gcd and not its degree, i.e. 300 calls.

In this section we will introduce a new algorithm called *SparseModGcd*, which is a sparse modular GCD algorithm. SparseModGcd takes advantage of *sparse interpolation* and has a better performance for polynomials which are sparse.

Again the input polynomials are $f_1, f_2 \in L[x]$, and g is their monic gcd. The problem is to find $\breve{g}$.

In [4], Wittkopf *et al.* presented a new algorithm called *LINZIP*, which is an extension to Zippel's algorithm (for gcd computation in $\mathbb{Z}[x_1, ..., x_n]$; see [16]) for the case where $\breve{g}$ is not monic in the main variable. When *LINZIP* uses sparse interpolation, it projects the input polynomials from $\mathbb{Z}[x_1, ..., x_n]$ to $\mathbb{Z}_p[x_1]$ by evaluating them at some evaluation points modulo a prime p. Similarly *SparseModGcd* projects the input polynomials from $\mathbb{Q}(t_1, ..., t_k)[z_1, ..., z_r][x]$ to $\mathbb{Z}_p[z_1, ..., z_r][x]$. Another difference between *LINZIP* and *SparseModGcd* is that, (as we discussed in Section 2.2.2) we need to deal with the zero divisors. Finally, similar to what is done in step 12 of subroutine P in *ModGcd*, we use univariate rational function reconstruction in *SparseModGcd* to recover the parameters $t_1, ..., t_k$.

2.3.1 Sparse Interpolation

The *sparse interpolation* algorithm first appeared in 1979 Ph.D. thesis of R. Zippel [16]. Later Wittkopf, Monagan and de Kleine (see [4]) extended the algorithm for the case where the gcd of the inputs is not monic. The main idea of using sparse interpolation in the modular gcd algorithm is that after finding the first image of the gcd in subroutine P or M(using dense interpolation) we know the *form* of the answer, i.e. we know (assuming the first image is of the correct form) the degree of g and which terms are present in g.

Example 2.19. Let $z = \sqrt{t}$. Consider the polynomials

$$f_1 = 3tx^2 + (-2t^2 + 3t)zx + 15x + 15z - 2t^3,$$

$$f_2 = 3tzx^2 + 15zx + (-2t^3 + 3t)x - 2zt^2 + 15.$$

For $p_1 = 7$, with the same method as Example 2.3, we get the first image $g_7 = tx + 4zt^2 + 5 \bmod 7$.

Now we assume that g_7 is of the correct form, that is, $g = atx + bzt^2 + c$, for some constants a, b, c. This will be true with high probability if p is large. Now we want to compute the gcd modulo a second prime $p_2 = 11$. First we choose $t = 2$ (again the actual choice for t needs to be random from a large set), evaluate the input polynomials at this point and run the Euclidean algorithm to get the first image

$$h_2 = \gcd(f_1(2, x), f_2(2, x)) \bmod 11 = x - 5z - 3.$$

If we evaluate the assumed form of the gcd g at $t = 2$ we will have

$$g(2, x) \bmod 11 = 2ax + 4bz + c.$$

Hence $2ax + 4bz + c \bmod 11 = x - 5z - 3$. From this we get the following linear system of equations

$$\{2a = 1, 4b = -5, c = -3\} \bmod 11.$$

Solving this system, we get $a = 6$, $b = 7$, $c = 8$ which means $\gcd(f_1, f_2) \bmod 11 = 6tx + 7zt2 + 8$. After making this monic, we obtain

$$g_{11} = tx + 3t^2z + 5 \bmod 11.$$

The reader may see that using a *dense* interpolation for computing the gcd modulo $p = 11$, requires at least three evaluation points, but using *sparse* interpolation needed only one evaluation point. This will improve performance when g is sparse and the improvement is multiplied for each variable.

Here is another example showing that we apply the sparse interpolation recursively when computing the first image of $\gcd(\check{f}_1, \check{f}_2) \bmod p_1$.

Example 2.20. Let $m(z) = z^2 - s - t$. Consider the following monic polynomials

$$f_1 = (x + s + 1)(x^3 + 10x^2z + x^2st^2 + 3xzs + 36xz + 17t),$$

$$f_2 = (x + t - s)(x^3 + 10x^2z + x^2st^2 + 3xzs + 36xz + 17t)$$

Suppose that we want to find the first image of $g = \gcd(f_1, f_2)$, modulo $p_1 = 7$. We take the first evaluation point to be $s = 1$ (note that these evaluation points must be chosen randomly in our algorithm, but here we choose small integers). By recursively calling the algorithm we get

$$g_1 = \gcd(f_1(1,t), f_2(1,t)) \bmod 7 = x^3 + t^2x^2 + 3zx^2 + 4xz + 3t.$$

Now we take the next evaluation point for s to be $s = 3$. Suppose $g_3 = \gcd(f_1(3,t), f_2(3,t)) \bmod 7$. From g_1 we know that, with high probability and assuming that g_1 is of the correct form, $g_3 = Ax^3 + Bt^2x^2 + Czx^2 + Dxz + Et$ for some constants A, B, C, D and E. Take the next evaluation point to be $t = 1$. By applying the Euclidean algorithm on $f_1(3,1)$ and $f_2(3,1)$ we get

$$g_{31} = x^3 + 3x^2 + 3zx^2 + 3xz + 3.$$

But we know that $g_{31} = g_3(t = 1)$, so we have the following equation

$$x^3 + 3x^2 + 3zx^2 + 3xz + 3 = Ax^3 + Bx^2 + Czx^2 + Dxz + E.$$

Solving these systems of equations,we obtain

$$A = 1, B = 3, C = 3, D = 3, E = 3.$$

This means $g_3 = \gcd(f_1(3,t), f_2(3,t)) = x^3 + 3t^2x^2 + 3zx^2 + 3xz + 3t$. After interpolating s (using dense interpolation), we get

$$h_1 = x^3 + st^2x^2 + 3zx^2 + (3s + 1)xz + 3t.$$

Now we apply the rational function reconstruction to the coefficients of h_1 but the output is h_1 and since $h_1 | f_1 \bmod 7$ and $h_1 | f_2 \bmod 7$ we conclude that $h_1 = \gcd(f_1, f_2) \bmod 7$. On the other hand $h_1 \nmid f_1$ and $h_1 \nmid f_2$ over $\mathbb{Q}$. So we need other images. This time we choose the next prime to be $p_2 = 11$. Let $h_2 = \gcd(f_1, f_2) \bmod 11$. From h_1, we know that (with high probability i.e. if h_1 is of the correct form)

$$h_2 = x^3 + a_1st^2x^2 + a_2zx^2 + (a_3s + a_4)zx + a_5t.$$

From the form of h_2, we need at least two images to solve for a_3 and a_4. Let our first set of evaluation points be $\alpha = (s = 1, t = 1)$. Applying the Euclidean algorithm on $f_1(\alpha)$ and $f_2(\alpha)$ will result in

$$g_{11} = x^3 + x^2 - zx^2 + 6xz + 6.$$

Again we know that $g_{11} = h_2(\alpha)$, hence we have

$$x^3 + x^2 - zx^2 + 6xz + 6 = x^3 + a_1x^2 + a_2zx^2 + (a_3 + a_4)xz + a_5.$$

Let's take our next evaluation point to be $\beta = (s = 2, t = 2)$. Again we apply Euclidean algorithm on $f_1(\beta)$ and $f_2(\beta)$ to obtain

$$g_{22} = x^3 + 8x^2 - zx^2 - 2xz + 6.$$

Solving the system of equations results in

$$a_1 = 1, a_2 = -1, a_3 = 3, a_4 = 3, a_5 = 6.$$

So $h_2 = x^3 + st^2x^2 - zx^2 + (3s + 3)xz + 6t$. Since $h_2 | f_1 \mod 11$ and $h_2 | f_2 \mod 11$, $h_2 = \gcd(f_1, f_2) \mod 11$. We can compute other images with the same method.

Remark 2.21. In general, we could evaluate at $t = \alpha_i \in \mathbb{Z}_p[z]/m(z)$. Instead we always evaluate at $t_i = \alpha_i \in \mathbb{Z}_p$ for two critical reasons. First, the linear system is over $\mathbb{Z}_p$ which means it can not run into zero divisors, which would further complicate the algorithm. Second, it means that we equate coefficients in z^ix^j instead of x^j. This reduces the number of images needed, hence the size of the linear systems to be solved.

Next we identify four classes of problems which may happen during sparse interpolation. These problems are *normalization problem*, *missing terms*, *unlucky contents* and *zero divisors*.

Normalization Problem

The first problem with sparse interpolation is called *normalization problem*. This problem happens when we are dealing with polynomials with a non-monic gcd in x. Consider the following example:

Example 2.22. Let $z = \sqrt{s + t}$. Suppose the input polynomials are

$$\check{f}_1 = (x - s + 1)\check{g} \text{ and } \check{f}_2 = (x + t + s)\check{g}$$

where

$$\check{g} = (15s + t)x^2 + 12s^2xz + 40st$$

is the gcd of $\check{f}_1$ and $\check{f}_2$. Suppose we have computed our first image modulo $p_1 = 7$ and obtained

$$\check{g}_7 = (s + t)x^2 + 5s^2xz + 5st.$$

So our assumed form of the actual gcd is

$$g_f = (As + Bt)x^2 + Cs^2xz + Dst$$

for some constants A, B, C and D. Now we want to compute the next image of gcd modulo $p_2 = 11$. Consider the evaluation point $\alpha = (s = 2, t = 1)$, we have

$$\check{g}_{21} = \gcd(\check{f}_1(2,1), \check{f}_2(2,1)) = x^2 + 9xz + 4.$$

The problem is that this image is *unique* up to a *scaling factor* m. That is

$$m(x^2 + 9xz + 4) = (2A + B)x^2 + 4Cxz + 2D,$$

But we do not know what m is. If we knew the leading coefficient of g, $\mathrm{lc}_x(\check{g}) = 15s + t$, then we could easily compute m

$$m = \mathrm{lc}_x(\check{g}(2,1)) \bmod 11 = 9.$$

Unfortunately there is no easy way of computing lc_g. An ingenious solution to the normalization problem, which we use in our algorithm, is presented by Wittkopf *et al.* in [4]. This solution does not require any factorization (which could be very expensive). The idea is to scale each image g_i with a *scaling factor* m_i. This introduces a new unknown variable, so we need some more images to construct consistent systems of linear equations. We give an example. Here, we follow the presentation of Wittkopf *et al.* but modify the example for L with $L = \mathbb{Q}(s,t)[z] \bmod m(z)$.

Example 2.23. Consider the input polynomials from the previous example. Our assumed form for the gcd is

$$g_f = (As + Bt)x^2 + Cs^2xz + Dst$$

for some constants A, B, C and D. Again we want to compute the next image of gcd modulo $p_2 = 11$. Remember the images of the gcd for evaluation point $\alpha = (s = 2, t = 1)$.

$$\check{g}_{21} = \gcd(\check{f}_1(2,1), \check{f}_2(2,1)) = x^2 + 9xz + 4.$$

Our first set of equations will be

$$m_1(x^2 + 9xz + 4) = (As + Bt)x^2 + Cs^2xz + Dst.$$

Here we have three equations $\{m_1 = 2A + B, 9m_1 = 4C, 4m_1 = 2D\}$ mod 11, and 5 unknown variables so we need another evaluation point. Consider the next evaluation point to be $\beta = (s = 1, t = 2)$. After applying the Euclidean algorithm on the inputs $\check{f}_1(\beta)$ and $\check{f}_2(\beta)$ we have

$$\check{g}_{12} = \gcd(\check{f}_1(1,2), \check{f}_2(1,2)) = x^2 + 2xz + 6.$$

So multiplying with the next scaling factor m_2 will result in

$$m_2(x^2 + 2xz + 6) = (As + Bt)x^2 + Cs^2xz + Dst.$$

Now we have the following system of linear equations:

$$\{2A + B = m_1, A + 2B = m_2, 4C = 9m_1, C = 2m_1, 2D = 4m_1, 2D = 6m_2\} \text{ mod } 11$$

It seems that we have 6 unknowns and 6 equations, but the last equation is a linear combination of the first five equations, so this system does not have a unique solution. Now we fix m_1 (the first multiplier) to be 1. Note that this can be done because the result we are seeking is only unique up to a scaling factor. Using this fact, our system of linear equations is now determined. The unique solution is

$$m_1 = 1, m_2 = 8, A = 9, B = 5, C = 5 \text{ and } D = 2.$$

Hence the image of gcd modulo $p_2 = 11$ will be

$$h = (9s + 5t)x^2 + 5s^2xz + 2st.$$

Since $h|\check{f}_1$ mod 11 and $h|\check{f}_2$ mod 11, we are done.

Here we ask how many images do we need to compute in order to have a determined system of linear equations.

Suppose the input polynomials are f_1 and f_2 such that $\gcd(\mathrm{lc}_x(\check{f}_1), \mathrm{lc}_x(\check{f}_2)) = 1$, i.e. the leading coefficients are relatively prime (this guarantees that $g = \gcd(f_1, f_2)$ is monic).

Suppose that our assumed form for the gcd is

$$g_f = \sum a_{ij}x^iz^j$$

and $a_{ij} \in \mathbb{Q}(t_1, ..., t_k)$. Let T be the number of terms in g_f. The number of images which is needed to construct a determined system of linear equations is the number of terms in the biggest (the one with more terms) coefficient among all a_{ij}'s.

This is quite obvious, since there is one system of equations for each coefficient a_{ij}, and the one for the biggest coefficient is the one with the most number of unknowns, so with this many equations, all the systems are guaranteed to be determined.

For the case where the gcd is non-monic, we use the scaling factors (multiple scaling case). Let n_i be the number of terms in the i'th coefficient of the assumed form and n_s be the sum over all n_i's. This time, after computing each image of the gcd we add d new equations to our system of linear equations (one equation for each term), but we also introduce a new unknown variable m_i which is the new scaling factor. For example, after computing i images, we will have $n_s + i - 1$ unknowns (there are originally n_s unknowns, and we have added one for each image except for the first scaling factor which is fixed to be 1). Hence in order to have a determined system, we need at least $\left\lceil \frac{n_s-1}{T-1} \right\rceil$ images.

This means that the worst case for this solution (adding the scaling factors) happens when $g = \gcd(f_1, f_2)$ has only a few terms (g is sparse in x, z and d is small) but each coefficient has a lot of terms.

In [4], Wittkopf *et al.* discuss the efficiency of the multiple scaling case. Consider the problem of finding a gcd which looks like

$$g_f = (a_2 s^2 + a_1 t + a_0)x^2 + (b_2 st^2 + b_1 sz + b_0)x + (c_1 t^2 + c_0 z).$$

The linear system of equations has a structure shown in Figure 2.1 where all entries not shown are zero. The solution can be easily computed by solution of a number of smaller subsystems corresponding to the rectangular blocks of non-zero entries augmented with the multiplier columns. With this method, the solution expense of the multiple scaling case is the same order as the single scaling case.

Bad form: Missing terms

The second problem which we may encounter during the use of sparse interpolation is called *missing terms.*

Suppose that our assumed form for the gcd is g_f.

Definition 2.24. A prime p is said to introduce *missing terms* if any term of g_f vanishes modulo p.

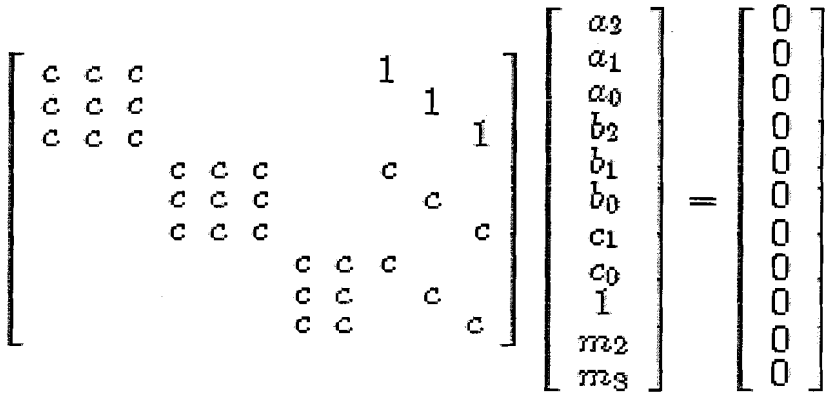

Figure 2.1: Linear system structure for the multiple scaling case, courtesy of Michael Monagan.

Definition 2.25. An evaluation point α is said to introduce *missing terms* if any coefficient of g_f vanishes at this evaluation point.

Unfortunately primes (or evaluation points) which introduce missing terms can not be avoided before computing the image. However, it is a good idea to impose that no term in the inputs $\check{f}_1, \check{f}_2$ should vanish modulo any of the primes. The problem with the missing terms is most important when we choose our assumed form based on an image which is computed modulo a prime (or at an evaluation point) which introduces a missing term.

Example 2.26. Let $m(z) = z^2 - s + t$. Consider the following input polynomials,

$$f_1 = (x + zs + t)g \text{ and } f_2 = (x^2 + 1)g$$

where

$$\check{g} = (t^2 - s + 1)x^3 + 70zx^2 + 13(t + 2).$$

is the gcd of f_1 and f_2.

Here the primes 2,5,7 and 13 introduce missing terms (the first three cause the second term to vanish and the last one makes the last term to vanish). The prime $p = 11$ does not introduce missing terms, but when we are computing the image of the gcd modulo this prime, $\alpha = (s = 8, t = 10)$ is not a good evaluation point since $\mathrm{lc}_g(8, 10) = 0$. Now suppose that we do not know what the gcd is, and we take our first prime to be $p_1 = 7$ (which is not a good choice). We get our first image

$$g_7 = (t^2 - s + 1)x^3 + 6t + 5.$$

From this image we take our assumed form for the gcd to be

$$g_f = (At^2 + Bs + C)x^3 + Ct + 6.$$

Now when we try to compute the gcd modulo $p = 11$, we will not get an image which is consistent with our assumed form with high probability. The inconsistency tells us that our assumed form is not of the correct form so we must restart the algorithm.

The probability of choosing a prime (or evaluation point) that makes a term of the gcd vanish is about T/p (T is the number of terms in the gcd). This means choosing larger primes will reduce the probability of having a missing term. In fact the primes should be much larger than the number of terms in the input polynomials.

Unlucky Content

Another problem which we have to avoid during sparse interpolation is an *unlucky content.*

Definition 2.27. (Wittkopf *et al.* [4]) For a polynomial $f = a_n x^n + ... + a_1 x + a_0$ with $a_i \in R$ for $0 \leq i \leq n$, the *content* $\text{cont}_x(f)$ is defined to be

$$\text{cont}_x(f) = \gcd(a_0, a_1, ..., a_n) \in R.$$

A prime p, is said to introduce an *unlucky content* if for two input polynomials f_1,f_2 with gcd $g = \gcd(f_1, f_2)$, $\text{cont}_x(g) = 1$ but $\text{cont}_x(g \bmod p) \neq 1$. Similarly an evaluation point $t = \alpha$ is said to introduce an unlucky content if $\text{cont}_x(g) = 1$ but $\text{cont}_x(g(\alpha)) \neq 1$.

Example 2.28. Consider the gcd $g = (12s + t)x + (s + 12t)$. We have

$$\text{cont}_x(g) = \gcd(12s + t, s + 12t) = 1.$$

But if we choose $p = 11$ we will obtain

$$\text{cont}_x(g \bmod p) = \text{cont}_x((s + t)x + (s + t)) = \gcd(s + t, s + t) = s + t$$

Hence $p = 11$ introduces an unlucky content and we must not use $p = 11$ to reconstruct the gcd. For any other prime $p \neq 11$ no content is present. The evaluation points $t = 0$ and $s = 0$ also introduce unlucky contents:

$$\text{cont}_x(g(0, t)) = \text{cont}_x(tx + 12t) = t,$$

$$\text{cont}_x(g(s, 0)) = \text{cont}_x(12st + s) = s.$$

If during sparse interpolation we choose our assumed form, g_f, based on an image which is computed modulo a prime (or evaluation point) which introduces an unlucky content, it is more likely that we will never get a system of linear equations which is determined and has a unique solution, no matter how many images we compute.

Example 2.29. Suppose we want to find the gcd of two polynomials f_1,f_2 with gcd $g = \gcd(f_1, f_2) = (10s + 7t)x^2 + 19sxz + 14st$. If we choose our first prime $p_1 = 7$, using the *dense interpolation* we get the first image

$$g_7 = x^2 + 4zx.$$

In fact $g \bmod 7 = sx^2 + 4szx$, but since our algorithm always returns the primitive associate of the gcd, it will remove $\text{cont}_x(g \bmod 7) = s$ and returns $x^2 + 4xz$. At this point we choose our assumed form to be (based on our images of the gcd modulo $p_1 = 7$)

$$g_f = Ax^2 + Bxz,$$

for some constants A and B. Now for our next image, we choose $p_2 = 11$. Consider the first evaluation point to be $\alpha = (s = 1, t = 1)$. After applying the Euclidean algorithm on $f_1(1,1)$ and $f_2(1,1)$ we get

$$h_1 = \gcd(f_1(1,1), f_2(1,1)) = x^2 + 5xz + 6.$$

But we can not equate $h_1 = g_f$ to solve for A and B. Since the degree of the new image is the same as the degree of the assumed form, the image is not unlucky so the assumed form must not be of the correct form. Now suppose we choose another evaluation point $\beta = (s = 2, t = 2)$. Again we apply Euclidean algorithm on $f_1(2,2)$ and $f_2(2,2)$ to get

$$h_2 = \gcd(f_1(2,2), f_2(2,2)) = x^2 + 5xz + 1$$

Again we have the same problem. In fact since $p_1 = 7$ introduced an unlucky content but $p_2 = 11$ does not, this problem always happens no matter how many evaluation points we choose. At this point we should throw away the assumed form and restart the algorithm since we do not know if it was the prime or a previous evaluation point that introduced the unlucky content.

As we discussed in the previous example, since the primes or evaluation points which introduce unlucky contents are rare, we do not detect the problem in advance. This is

because the detection of such primes and evaluation points could be expensive and create a bottleneck for the algorithm. We will detect an unlucky content by checking if $f_1(\alpha, t_i, ..., t_k, x) \bmod p$ and $f_2(\alpha, t_i, ..., t_k, x) \bmod p$ have a common content in $\mathbb{Z}[t_i]$ where $\alpha = (t_1 = \alpha_1, ..., t_{i-1} = \alpha_{i-1})$ is the evaluation point. If they do, the algorithm discards p and chooses another prime (this is because we don't know if it is p or α which introduced the unlucky content). Note that with this method, the algorithm only computes univariate contents to detect the unlucky content. This is because the unlucky content will eventually be reduced to a univariate content as we evaluate the parameters one by one.

Zero Divisors

The next problem which may happen during the sparse interpolation is that when we are trying to compute univariate gcds, the Euclidean algorithm could hit a zero divisor in which case it fails. See Example 2.14.

2.3.2 Algorithm SparseModGcd

We now present the SparseModGcd algorithm. This modular GCD algorithm first calls subroutine M which computes the GCD in $L[x]$ from a number of images in $L_p[x]$. Subroutine P which is called by subroutine M computes the GCD in $L_p[x]$ using both dense and sparse interpolations. Finally subroutine S, which stands for Sparse Interpolation and is called by subroutine P, does the sparse Interpolation.

Algorithm SparseModGcd

Input: $f_1, f_2 \in L[x]$ and $m_1, ..., m_r \in F[z_1, ..., z_r]$ where $F = \mathbb{Q}(t_1, ..., t_k)$ s.t. $\text{cont}_x(g) = 1$.
Output: $\check{g}$, where g is the monic gcd of f_1 and f_2 in $L[x]$.

1. Call Subroutine M with input $\check{f}_1$, $\check{f}_2$ and $\check{m}_1$,...,$\check{m}_r$.

Subroutine M

Input: $f_1, f_2 \in D[z_1, ...z_r]/\langle m_1, .., m_r\rangle [x]$ and $m_1, ..., m_r \in D[z_1, ...z_r]$ were $D = \mathbb{Z}[t_1, ..., t_k]$.
Output: $\check{g}$, where g is the monic gcd of f_1 and f_2.

1. Set $n = 1, G = 0$, $form = 0$.
2. Main Loop: Take a new prime p_n.

3. Check if p_n is a bad prime, if it is go back to step 2.
4. Let $g_n \in D_{p_n}[z_1, .., z_r, x]$ be the output of subroutine P applied to f_1, f_2, form, m_1 mod p ,..., m_r mod p. If g_n = "contfailed" then go back to step 2.
5. If g_n = "ZeroDivisor" or g_n = "Unlucky" then go back to step 2.
6. If g_n = "Bad_Form" then go back to step 1.
7. If $g_n = 1$ then return 1.
8. If $G = 0$ then set $G = g_n$ and $m_c = p$ then go to step 12.
9. If $\deg_x(g_n) < \deg_x(G)$ then set $G = g_n$, $m_c = p$, $form = g_n$ then go to step 11. /*All previous primes where unlucky */
10. If $\deg_x(g_n) > \deg_x(G)$ then go back to step 2. /* p_n is an unlucky prime */
11. Combine the images $\{g_1, ..., g_n\}$ using Chinese remaindering to obtain G mod m_c.
12. Set $n = n + 1$.
13. Apply integer rational reconstruction to obtain h from G mod m_c. If this fails, Set $form = g_n$ then go back to step 2.
14. Clear fractions in $\mathbb{Q}$: Set $h = \check{h}$.
15. Set $form = g_n$.
16. Trial division: if $h|f_1$ and $h|f_2$ then return h, otherwise, go back to step 2.

Subroutine P

Input: f_1,f_2,$form \in D_p[z_1, ...z_r]/\langle m_1, .., m_r\rangle\,[x]$ and $m_1, ..., m_r \in D_p[z_1, ...z_r]$.
Output: Either $\check{g}$ or "ZeroDivisor" or "Unlucky" or "Bad_Form" or "contfailed" if the algorithm fails to compute the primitive associate of the monic gcd of f_1 and f_2 because of the bad choice of the prime or evaluation point.

0. If the GCD of the inputs has a content in t_k then return "contfailed". /* There is an unlucky content */
1. If k (The number of parameters) = 0 then output the result of the Euclidean algorithm applied to f_1,f_2.
2. If $form \neq 0$ then go to step 28.
3. Set $n = 1, d = 1, G = 0$.
4. Take a new evaluation point α_n at random from $\mathbb{Z}_p$.
5. Check if α_n is a bad evaluation point, if it is go back to step 4.
6. Let $g_n \in \mathbb{Z}_p[t_1, .., t_{k-1}][z_1, .., z_r, x]$ be the output of subroutine P applied to f_1,f_2,m_1, ..., m_r at $t_k = \alpha_n$ and $form = 0$. If g_n = "contfailed" then return "contfailed".

7. If g_n = "ZeroDivisor" then return "ZeroDivisor".
8. If g_n = "Unlucky" then go back to step 4. / *Unlucky evaluation point */
9. If g_n = "failed" or "ZeroDivisor" then
   9.1. Set $d = d + 1$.
   9.2. If $d > n$ output "ZeroDivisor", else go back to step 4.
10. If $g_n = 1$ then return 1.
11. Set $form = g_n$, $n = n + 1$.
12. Main Loop: Take a new evaluation point α_n at random from $\mathbb{Z}_p$.
13. Check if α_n is a bad evaluation point. If it is go back to step 12.
14. Let $g_n \in \mathbb{Z}_p[t_1, .., t_{k-1}][z_1, .., z_r, x]$ be the output of subroutine S applied to f_1,f_2,$m_1, ..., m_r$ at $t_k = \alpha_n$ and form.
15. If g_n = "Bad_Form" then go back to step 3. /*Our assumed form for gcd is not of the correct form */
16. If g_n = "Unlucky" then go back to step 12. /* Unlucky evaluation point */
17. If g_n ="failed" or "ZeroDivisor" then
   17.1. Set $d = d + 1$.
   17.2. If $d > n$ output "ZeroDivisor", else go back to step 12.
18. Set $g_n = \text{monic}(g_n)$.
19. If $g_n = 1$ then return 1.
20. If $G = 0$ then set $G = g_n$ and $m_c = t_k - \alpha_n$ then go to step 25.
21. If $\deg_x(g_n) < \deg_x(G)$ then set $G = g_n$, $m_c = t_k - \alpha_n$, $form = g_n$ then go to step 25. /*All previous evaluation points where unlucky */
22. If $\deg_x(g_n) > \deg_x(G)$ then go back to step 12. /* α_n is an unlucky evaluation point */
23. Chinese remainder $\{g_1, ..., g_n\}$ to obtain G mod $m_c(t_k)$.
24. Set $n = n + 1$.
25. Apply rational function reconstruction to the coefficients of G to obtain $h \in \mathbb{Z}_p(t_k)$ $[t_1, ..., t_{k-1}]$ $[z_1, ..., z_r, x]$ s.t. $h \equiv G \mod m_c(t_k)$. If this fails, go back to step 12.
26. Clear fractions in $\mathbb{Z}_p(t_k)$: Set $h = \check{h}$.
27. Trial division: if $h|f_1$ and $h|f_2$ then return h, otherwise, go back to step 12.
28. Let $g_n \in \mathbb{Z}_p[t_1, .., t_k][z_1, .., z_r, x]$ be the output of subroutine S applied to f_1,f_2,$m_1, ..., m_r$ and form. /*At this step, we already know the form of the gcd */

29. If g_n = "Bad_Form" then return "Bad_Form". /*Our assumed form for the gcd is not of the correct form */
30. If g_n = "Unlucky" or g_n = "ZeroDivisor" then return g_n.
31. return $monic(g_n)$.

Subroutine S

Input: f_1,f_2,$form \in D_p[z_1,...z_r]/\ \langle m_1,..,m_r\rangle\ [x]$ and $m_1,...,m_r \in D_p[z_1,...z_r]$ where $D_p = \mathbb{Z}_p[t_1,...,t_k]$.

Output: Either $\check{g}$ or "Bad_Form" or "ZeroDivisor" or "Unlucky" if the algorithm fails to compute the primitive associate of the monic gcd of f_1 and f_2.

1. If k (the number of parameters) = 0 then output the result of the Euclidean algorithm applied to f_1,f_2.
2. Set $C = \text{coeffs}_{x,z_1,...,z_r}(form)$, $T = \text{monomials}_{x,z_1,...,z_r}(form)$. /*$C$ is the list of all coefficients of form in D_p and T is the list of all monomials s.t. $f = \Sigma(C_iT_i)$ */
3. Set U to be the minimum number of images needed. /* This is based on what we discussed in Section 2.3.2 */
4. Set $z = 0$, $u = 0$.
5. for i from 1 to U do
   - 5.1. Take a new random evaluation point $\alpha_i = (t_1 = a_1,...,t_k = a_k)$ in $\mathbb{Z}_p^k$ which is not bad.
   - 5.2. Let g_i be the output of the Euclidean algorithm applied to $f_1(\alpha_i)$, $f_2(\alpha_i)$, $m_1(\alpha_i)$, ..., $m_r(\alpha_i)$.
   - 5.3. If g_i = "failed" then
     - 5.3.1. Set $z = z + 1$.
     - 5.3.2. If $z > i$ then return "ZeroDivisor" else go back to step 5.1.
   - 5.4. If $\deg_x(g_i) > \deg_x(form)$ then
     - 5.4.1. Set $u = u + 1$.
     - 5.4.2. If $u > i$ then return "Unlucky" else go back to step 5.1.
   - 5.5. If $\deg_x(g_i) < \deg_x(form)$ then return "Bad_Form".
   - 5.6. If the number of terms in g_i with respect to to $x, z_1, ..., z_r$ is greater than the number of terms in the assumed form then return "Bad_Form". /* Missing terms in the assumed form*/

6. Construct the system of linear equations based on the g_i's (images), α_i's (evaluation points), and C, and solve it.
7. If the system is inconsistent, return "Bad_Form".
8. Construct and return g_p ($g \mod p$), using the solution from the system of equations and T (Terms in the assumed form).

We now describe how SparseModGcd deals with three main problems, namely *bad primes and evaluations*, *unlucky primes and evaluations*, *unlucky contents* and *missing terms*.

Note that the *zero divisors problem* is treated in the same way as in ModGcd algorithm (see Section 2.2.5).

Bad Primes and Bad Evaluation Points

As we stated before in Section 2.2.1, bad primes and bad evaluation points must not be used during the gcd computation. Fortunately there is an easy way of finding whether or not a prime p (or an evaluation point α) is bad. In subroutine M of SparseModGcd algorithm, we avoid bad primes, by testing if the new prime p_n divides the leading coefficient of any of $f_1, f_2, m_1, ..., m_r$. If it does, then p_n is a bad prime.

For the case of bad evaluation points, in subroutine P (for both dense and sparse interpolation parts) we discard bad evaluation points by simply testing if the leading coefficient of any of $f_1, f_2, m_1, ..., m_r$ vanishes after evaluating at the new evaluation point α_n.

Unlucky Primes and Evaluation Points

As we discussed before, unlucky primes and evaluation points, cause the new image to have a higher degree in the main variable x compared to g. Unfortunately there is no way of checking if a prime (or an evaluation point) is unlucky before computing the image of the gcd.

During the sparse interpolation, we evaluate the inputs f_1 and f_2 at a sequence of evaluation points, then we interpolate the last parameter, using a dense method. At this point we have our first image from which we choose the assumed form. If this image is based on an unlucky evaluation point, subroutine S will eventually get an image with lower degree in x than the assumed form. In this case the sparse interpolation fails by returning "Bad_Form" and the algorithm restarts. If the first image is based on an unlucky prime, subroutine P does not detect it and returns an image with a high degree. But in this case,

subroutine M will choose the new assumed form based on this bad image and hence the algorithm will eventually detect it during the sparse interpolation by computing an image with lower degree. At this point we assume that the first image is of the correct form. If during the interpolation of other variables using the sparse interpolation algorithm we encounter an image which has a higher degree than the assumed form, it can be because of the choice of the evaluation points in either subroutines P or S. In Step 5.4 of the algorithm, we count the number of times which we encounter an unlucky image. If this happens a lot, we conclude that the evaluation point in subroutine P is unlucky otherwise we assume that the current evaluation point is unlucky so we choose another one.

Finally, after we compute the image of the gcd modulo the first prime which is not unlucky, if we choose a new unlucky prime, we will get unlucky images during the sparse interpolation so subroutine M simply chooses another prime. Since there are finitely many unlucky primes, subroutine M will eventually compute some good images.

Unlucky Contents

As we described in Section 2.3.1, the single variable content check in Step 0 of subroutine P will eventually detect an unlucky content. If the check in Step 0 detects an unlucky content, subroutine P will fail all the way up to subroutine M which then throws away the current prime and starts with another one.

This strategy which is first introduced in [4] is efficient since from the point where an unlucky content is introduced to the point where it is detected only some variable evaluations and single variable content checks have been performed.

Missing Terms

If the algorithm chooses the assumed form of the gcd based on an image with missing terms, subroutine S will eventually get an image with more terms than the assumed form. In this case (step 5.6 of subroutine S) the algorithm restarts (step 15 of subroutine P) to find a new assumed form.

Chapter 3

Implementation

We have implemented algorithm SparseModGcd in Maple 10 (see Appendix A). In this chapter we will first describe the bottlenecks in the implementation of SparseModGcd algorithm and finally we present a running time comparison of SparseModGcd and ModGcd algorithms.

3.1 Bottlenecks

In this section we will discuss some bottlenecks in the implementation of our SparseModGcd algorithm namely *trial division*, *rational function reconstruction*, *sparse interpolation* and *univariate gcd computation*. We will start with an example. In our example Maple is using 31.5 bit primes on a 64 bit machine.

Example 3.1. Let

$$f_1 = 2t^2x^5 + 2zt^2x^4 + 5zy^2x^3 + 5z^2y^2x^2 + \left(-3tz^2y + 1\right)x - 6t^2y + z,$$

$$f_2 = 2x^6t^2 + \left(2t^2 + 5zy^2\right)x^4 + \left(5zy^2 - 3tz^2y + 1\right)x^2 - 3tz^2y + 1$$

be input polynomials and $m(z) = z^3 - 2t$. The output of our program is as follows.

```
Entering MGCD...
Calling PGCD...Current prime is :  3037000453
   y=2926416935
   t=198304613
   t=640439653
```

```
   t=417177802
   t=2984266210
   t=537655880
   t=1361931892
RFR(t) succeeded.  Number of points used is 5
Assumed form for the gcd is computed.  Monomials are [1, tz^2, zx^2, t^2x^4]
   y=2107063881
      Sparse Interpolation...succeeded.
   y=1884392679
      Sparse Interpolation...succeeded.
   y=902841101
      Sparse Interpolation...succeeded.
   y=802611721
      Sparse Interpolation...succeeded.
RFR(y) succeeded.  Number of points used is 4
PGCD succeeded.
Integer Rational Reconstruction...succeeded.
Division Check...succeeded...gcd found !
                                   2t^2x^4 + 5zy^2x^2 - 3tz^2y + 1
Timings:
Total time:  0.057 (in CPU seconds)
Trial Division:  6.78%, Rational Function Reconstruction:  3.39%
Sparse Interpolation:  5.08%, Univariate GCD computation:  67.79%
```

As you see, more than eighty percent of the total running time is spent on the four specified bottlenecks. And this is after our optimizations.

3.1.1 Trial Division

Recall from Section 2.2.5 that trial division is used to check whether we have computed the correct image of the gcd at the end of subroutine P. Suppose that a is the average number of times which the trial division routine is called for computing g_p the image of gcd modulo p. Let q be the probability that the trial division succeeds and t_d be the average time for doing one trial division.

Unfortunately we can not decrease t_d, i.e. we can not improve the trial division algorithm very much, but we can decrease a by increasing q. This means that we will not do the trial

division unless we know with high probability it will succeed. This can be accomplished by reducing the number of times that the rational function reconstruction method (in Step 25 of subroutine P) succeeds but does not output g.

We achieve this by using a slightly different method for rational function reconstruction which is described in the next section. Now, the average number of times which the trial division routine is called, is very close to 1.

3.1.2 Rational Function Reconstruction

As we discussed in the previous section, we need to do the *rational function reconstruction* in such a way that when it succeeds, its output (after clearing the fractions) is the gcd, g and hence will divide both of the input polynomials f_1 and f_2 modulo the prime p. For this, we use the *Maximal Quotient Rational Reconstruction* method which is presented by Monagan in [8].

The idea is to use more (one more) evaluation points than are necessary to reconstruct $n/d \in F(y)$.

Example 3.2. Suppose we have $f \in \mathbb{Z}_{23}[y]$ such that

$$f(1) = 9, f(2) = 19, f(3) = 5, f(4) = 18, f(5) = 11, f(6) = 2.$$

We want to find a rational function $n/d \in \mathbb{Z}_{23}(y)$ such that

$$\frac{n(\alpha)}{d(\alpha)} = f(\alpha),\ d(\alpha) \neq 0,\ \alpha \in \{1,2,3,4,5,6\}.$$

Using polynomial interpolation we can easily compute $u = 5y^5+11y^4+22y^3+5y^2+4y+8$ satisfying $u(\alpha) = f(\alpha)$ on these six points. Let

$$m = \prod_{\alpha_i \in \{1,2,3,4,5,6\}} (y - \alpha_i) \bmod p = y^6 + 2y^5 + 14y^4 + y^3 + 14y^2 + 7y + 7.$$

After applying the extended Euclidean algorithm as described in Section 1.4 to inputs m and u, we get the following set of solutions

$$\left\{5y^5 + 11y^4 + 22y^3 + 5y^2 + 4y + 8, \frac{5y^4 + 14y^3 + 2y^2 + 9y + 14}{y + 9}, \frac{13y^3 + 12y^2 + 2}{y^2 + 13y + 12}\right.$$

$$\left.\frac{6y^2 + 20y + 13}{y^3 + 17y^2 + 20y + 20}, \frac{22y + 22}{y^4 + 14y^3 + 11y^2 + 7}, \frac{19}{y^5 + y^4 + 17y^3 + 17y^2 + 18y + 12}\right\}.$$

All have degree 5, so rational function reconstruction fails. As we mentioned before, maximal quotient rational reconstruction method uses one more evaluation point to select $\frac{n}{d}$. Assuming $u(7) = 14$ we have

$$m = \prod_{\alpha_i \in \{1,2,3,4,5,6,7\}} (y - \alpha_i) \bmod p = y^7 + 18y^6 + 18y^4 + 7y^3 + y^2 + 4y + 20,$$

$$u = y^6 + 7y^5 + 2y^4 + 19y^2 + 11y + 15.$$

Again using the extended Euclidean algorithm we get the following set of solutions

$$\left\{ \frac{13y^3 + 12y^2 + 2}{y^2 + 13y + 12}, y^6 + 7y^5 + 2y^4 + 19y^2 + 11y + 15, \frac{10y^5 + 4y^4 + 12y^3 + 12y^2 + 17y + 7}{y + 11}, \right.$$

$$\frac{22y^2 + 11y + 16}{y^4 + 10y^3 + 12y^2 + 12y + 19}, \frac{18y + 14}{y^5 + 6y^4 + 18y^3 + 3y^2 + 18y + 1},$$

$$\left. \frac{3}{y^6 + 7y^5 + 19y^4 + 17y^3 + 7y^2 + 5y + 21} \right\}.$$

One can see that $\frac{n}{d} = \frac{13y^3+12y^2+2}{y^2+13y+12}$ is the only solution from the above set with $\deg(n) + \deg(u) = 5$, hence it is the output of maximal quotient rational reconstruction method.

We refer the reader to Monagan [8] for a detailed description of the Maximal Quotient Rational Reconstruction algorithm.

3.1.3 Sparse Interpolation

Sparse interpolation is another part which could slow down the algorithm significantly if it is not implemented properly.

The most time consuming part of the sparse interpolation routine is at Step 5.2 where the algorithm *evaluates* the input polynomials f_1 and f_2 at the new evaluation point α_i. To overcome this, we first form a matrix which includes all the coefficients of both input polynomials f_1 and f_2 with respect to the main variable x and $z_1, ..., z_r$. Next we use a modular method (which is part of the LinearAlgebra package in Maple 10 and is coded in C for machine primes) to evaluate each entry of the matrix at the new evaluation point α_i. Finally, using the evaluated matrix, we can easily form $f_1(\alpha)$ and $f_2(\alpha)$.

3.1.4 Univariate Gcd Computation

Suppose that we want to compute $g_p = \gcd(f_1, f_2) \bmod p$ using the sparse interpolation. This involves some univariate gcd computations in $R[x]$ where $R = L(t_1 = \alpha_1, ..., t_k = \alpha_k)/\langle m_1, ..., m_r \rangle$. Thus R mod p is a finite ring with r extensions. As we discussed in Section 2.3.1, the number of univariate images we need to compute is at least $n_p = \lceil (n_s - 1)/(T - 1) \rceil$ where n_s is the number of unknown variables in our assumed form for the gcd and T is the number of terms in the assumed form. If n_p is a large number, then the *Euclidean algorithm* (which is used for univariate gcd computation) could take a large amount of time. Therefore the Euclidean algorithm should be implemented efficiently (it needs to be coded in C).

In our implementation of the SparseModGcd algorithm, we use a version of Euclidean algorithm which is designed for polynomials over a ring R (Monagan and van Hoeij [9]). The cost of the Euclidean algorithm is $O(n^2D^2) \times N$ as implemented. Here D is the degree of the algebraic function field L, n is the degree of g in the main variable and N is the number of times that the Euclidean algorithm is called.

3.2 Benchmarks

We have compared the Maple implementations of SparseModGcd and ModGcd on three problem sets. The first two sets consist of input polynomials f_1 and f_2 with a sparse gcd $g = \gcd(f_1, f_2)$. In contrast, each pair of polynomials in the third set has a rather dense gcd. There is only one field extension available in these problem sets. Next, we present some timings for SparseModGcd algorithm on a problem set with two field extensions ($r = 2$). The purpose of the last benchmark is to count the number of bad and unlucky primes and evaluation points, zero divisors, unlucky contents and missing terms that SparseModGcd encounters for two random polynomials. All the timings in this section are in CPU seconds and obtained using Maple 10 on a 64 bit AMD Opteron CPU running Linux using 31.5 bit primes.

As we discussed in Chapter 2, SparseModGcd is expected to have a better performance than ModGcd on the first two sets of problems.

SPARSE-1

Let

$$m(z) = z^3 - sz^2 - t^2z - 5 - 3u,$$

$$g = sx_1^n + tx_2^n + ux_3^n + \sum_{j=1}^{3}\sum_{i=0}^{n-1} r_{ij}^{(1)} z^{j-1}x_j^i + \sum_{w=[s,t,u]}\sum_{k=0}^{n} r_{w_k}^{(1)} w^k,$$

$$a = tx_1^n + ux_2^n + sx_3^n + \sum_{j=1}^{3}\sum_{i=0}^{n-1} r_{ij}^{(2)} z^{j-1}x_j^i + \sum_{w=[s,t,u]}\sum_{k=0}^{n} r_{w_k}^{(2)} w^k,$$

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

where each $r_{jk}^{(i)}$ is a positive random number less than 100. For $n = 1, 2, ..., 10$, let $f_1 = a \times g$ and $f_2 = b \times g$. Thus we have 10 gcd problems, all with one field extension $m(z)$, three parameters s, t and u and three variables x_1, x_2 and x_3. Each input polynomial is of degree $2n$ in each variable x_1, x_2, x_3 and the gcd $g = \gcd(f_1, f_2)$ is of degree n in each variable.

n	SparseModGcd	ModGcd
1	0.170	0.50
2	0.359	2.30
3	0.662	7.94
4	1.164	23.57
5	1.868	60.54
6	2.938	139.9
7	4.476	301.58
8	6.512	602.765
9	9.187	> 2000
10	12.611	NA

Table 3.1: Timings (in CPU seconds) of SparseModGcd compared to ModGcd on the first set of problems SPARSE-1 (NA means not attempted)

Table 3.1 shows the running time comparison between SparseModGcd and ModGcd algorithms. Since the gcd g in this case is very sparse (g has $6n + 3$ terms and $\deg(g) = n$ in any of x_1, x_2 and x_3), a better performance is expected from SparseModGcd. The data demonstrates this clearly.

SPARSE-2

Let

$$m(z) = z^3 - (s+r)z^2 - (t+v)^2 z - 5 - 3u,$$

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

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

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

This problem is similar to the previous problem set. Each $r_{j_k}^{(i)}$ is a positive random number less than 100. For $n = 1, 2, ..., 10$ with $f_1 = a \times g$ and $f_2 = b \times g$, we have 10 gcd problems, all with one field extension $m(z)$. This time there are five parameters r, s, t, u and v and four variables x_1, x_2, x_3 and x_4. Each input polynomial is of degree $2n$ in the first three variables and $2n - 2$ in x_4.

n	SparseModGcd	ModGcd
1	0.40	8.70
2	1.29	114.78
3	2.40	879.26
4	4.46	> 2000
5	7.57	NA
6	12.51	NA
7	20.25	NA
8	29.73	NA
9	43.03	NA
10	61.87	NA

Table 3.2: Timings (in CPU seconds) of SparseModGcd compared to ModGcd on the second set of problems SPARSE-2 (NA means not attempted)

Table 3.2 illustrates the running time comparison of the two algorithms. Again SparseModGcd has a better performance compared to ModGcd, which is expected because $g =$

$\gcd(f_1, f_2)$ is sparse. But since there are more variables and more parameters in this example, the exponential running time of ModGcd results in it being even less competitive with SparseModGcd.

DENSE-1

Let $m(z) = z^2 - sz - 3$. Suppose g, a and b are three randomly chosen polynomials in variables x_1, x_2, s and z of total degree n which are dense. That is the term $x_1^{d_1} x_2^{d_2} s^{d_3} z^{d_4}$ with $d_1 + d_2 + d_3 + d_4 \leq n$ is present in each of these three polynomials. This means that each of them has exactly $\sum_{i=0}^{n} \binom{n+4}{4}$ number of terms.

For $n = 1, 2, ..., 10, 15$, let $f_1 = g \times a$ and $f_2 = g \times b$. Since in this set of problems the gcd g is dense, ModGcd algorithm is expected to perform better.

n	SparseModGcd	ModGcd
1	0.033	0.029
2	0.072	0.058
3	0.151	0.141
4	0.313	0.307
5	0.498	0.557
6	0.921	1.272
7	1.584	2.091
8	2.527	3.244
9	4.191	5.024
10	7.704	7.437
15	62.758	50.228

Table 3.3: Timings (in CPU seconds) of SparseModGcd compared to ModGcd on the fourth set of problems DENSE-1

Table 3.3 shows the running times of both ModGcd and SparseModGcd algorithms for this set of problems.

SPARSE-3

In this problem set the input polynomials are the same as SPARSE-1, but there are two field extensions $m_1(z_1) = z_1^3 - sz_1^2 - t^2 z_1 - 5 - 3u$ and $m_2(z_2) = z_2^2 - z_2 z_1 + sz_2 + 3t - u$.

The timings for SparseModGcd algorithm on this problem set are shown in Table 3.4.

n	SparseModGcd
1	0.592
2	1.577
3	3.651
4	7.386
5	13.76
6	23.65
7	38.79
8	59.77
9	90.03
10	129.14

Table 3.4: Timings (in CPU seconds) of SparseModGcd compared to ModGcd on the fifth set of problems SPARSE-3

SPARSE-4

The minimal polynomials in this set of problems are the same as SPARSE-3. Let

$$g = sx_1^n + tx_2^n + ux_3^n + P_1,$$

$$a = tx_1^n + ux_2^n + sx_3^n + P_2,$$

$$b = ux_1^n + sx_2^n + tx_3^n + P_3.$$

Here P_1, P_2 and P_3 are three random polynomials in $\{x_1, x_2, x_3, s, t, u, z_1, z_2\}$. For $n = 1, 2, ..., 10$, let $f_1 = a \times g$ and $f_2 = b \times g$. We have run SparseModGcd algorithm 100 times for each n and we have encountered no bad primes, no bad evaluation points, no unlucky primes, no unlucky evaluation points, no zero divisors, no unlucky contents and no missing terms. We used this command in Maple 10:

```
> m1 := z1^3-s*z1^2-t^2*z1-5-3*u;
> m2 := z2^2-z2*z1+s*z2+3*t-u;
> g := s*x1^n+t*x2^n+u*x3^n+randpoly([x1,x2,x3,s,t,u,z1,z2], terms=50, degree=n);
> a := t*x1^n+u*x2^n+s*x3^n+randpoly([x1,x2,x3,s,t,u,z1,z2], terms=50, degree=n);
> b := u*x1^n+s*x2^n+t*x3^n+randpoly([x1,x2,x3,s,t,u,z1,z2], terms=50, degree=n);
> f1 := expand(g*a); f2 := expand(g*b);
```

So P_1, P_2 and P_3 have two digits random integer coefficients.

Chapter 4

Summary

We have designed and implemented SparseModGcd, a sparse modular GCD algorithm for polynomials over algebraic function fields based on ModGcd algorithm which is presented by Monagan, van Hoeij in [10]. In contrast to ModGcd, SparseModGcd uses Zippel's sparse interpolation algorithm, so it is much more efficient for polynomials with a sparse gcd. Moreover it can be used in the case where there are multiple field extensions to the algebraic function field.

ModGcd and accordingly SparseModGcd are extensions of the modular GCD algorithm of Brown for $\mathbb{Z}[x_1, ..., x_n]$ and Encarnacion for $\mathbb{Q}(\alpha)[x]$ to function fields. In these algorithms, we first try to find some images of the gcd modulo a series of prime numbers using an interpolation algorithm, and then apply the Chinese remainder theorem to compute the actual gcd. ModGcd uses dense interpolation but SparseModGcd uses both dense interpolation (for finding the first image) and sparse interpolation. As a result, SparseModGcd has a better performance when $g = \gcd(f_1, f_2)$ is sparse. Zippel's sparse interpolation however only works when the gcd is monic, i.e., when the leading coefficient of the gcd is 1. To overcome this problem, we use the multiple scaling factors idea, which is presented by Wittkopf et al. in [4]. Furthermore, to speed up the algorithm, our implementation uses Monagan's maximal quotient rational function reconstruction.

Finally, to demonstrate the efficiency of our algorithm, we have compared the Maple implementation of ModGcd with our Maple implementation of SparseModGcd on three problem sets. As expected, SparseModGcd turned out to have a much better performance on the problem sets which contain two input polynomials with a sparse gcd.

One future extension point is to parallelize the SparseModGcd algorithm, so that it can

compute two or more images simultaneously. This could increase the performance of the algorithm significantly, since in most cases more than sixty percent of the running time of the algorithm is spent on univariate gcd computation.

Another improvement to our algorithm is to speed up the implementation of the Euclidean algorithm by coding it in C.

Finally, we are planning to put the implementation of SparseModGcd in the next release of Maple.

Appendix A

Maple Implementation of SparseModGcd algorithm

```
# Code is available on http://www.sfu.ca/~sjavadi/SparseModGcd/

# Input:  f1,f2.
# Output:  ğ, where g is the monic gcd of f1 and f2.
GCD := proc(f1 , f2)
    local R,ml,zl,g,ff1,ff2,i,lR;
    R := [op(indets([f1,f2],RootOf))];
    if nops(R) = 0 then return primpart(gcd(f1 , f2)); fi;
    R := sort(R, proc(x,y) evalb(length(x) < length(y)) end proc);
    ml , zl := RootConvert(R , 1);
    ff1 := f1;
    ff2 := f2;
    lR := R;
    for i from 1 to nops(lR) do
        ff1 := SUBS(R[i] = zl[i] , ff1);
        ff2 := SUBS(R[i] = zl[i] , ff2);
        R := SUBS(R[i] = zl[i] , R);
    od;
    g := ModGcd(expand(ff1),expand(ff2),ml,zl);
    if g = "failed" then return g; fi;
    g := RConvert(g , zl , lR);
    return g;
```

```
end:
macro(
    EVAL_SUBS = evalsubs,
    MATRIX_SOLVE = mSolve,
    MCHREM = mgcd_chrem,
    I_RATRECON = n_iratrecon,
    MDIV = MTrialDivision,
    CONTENT = mContent,
    EVAL = Phi,
    PCHREM = pgcd_chrem,
    RATRECON = nRatrecon,
    PDIV = PTrialDivision,
    EUCLID = euclidean,
    SUBS = subs
):
# Uses recden package.
# See http://www.cecm.sfu.ca/CAG/code/NGCD/recden
euclidean := proc (f1, f2, vl, p, ml)
    local r1 , r2 , g , tt;
    r1 := rpoly(f1 , vl , ml, p);
    r2 := rpoly(f2 , vl , ml, p);
    g := traperror(gcdrpoly(r1 , r2));
    if g=lasterror then return "failed" fi;
    g := rpoly(g);
    return g;
end:
evalsubs := module()
    option package;
    export initialize, substitute;
    local MA,MB,CA,CB,NA,NB,RA,RB,CML,NM,NML,MR;
    initialize := proc(a,b,vzl,zl,dat_t,ml,p)
        local MNM;
        CA:=[coeffs(a,vzl,'MA')];MA:=[MA];NA:=nops(CA);CA:=Vector(CA);
        CB:=[coeffs(b,vzl,'MB')];MB:=[MB];NB:=nops(CB);CB:=Vector(CB);
        RA := Create(p, NA, 0, 0, dat_t); RB := Create(p, NB, 0, 0, dat_t);
        CML:=map(proc(m) cm:=coeffs(m,zl,'MM'); return [[cm],[MM]]; end,ml);
        NM := nops(ml);
```

```
        NML := map(proc(m) nops(m[1]) end proc , CML);
        MNM := max(op(NML));
        MR := Create(p,NM,MNM,0,dat_t);
    end:
    substitute := proc(tt,p)
        local nml,na,nb,i;
        Mod(p,CA,tt,RA);
        Mod(p,CB,tt,RB);
        na := add(RA[i]*MA[i] , i = 1..NA);
        nb := add(RB[i]*MB[i], i = 1..NB);
        nml := [];
        Mod(p,CML[1..NM,1],tt,MR);
        for i from 1 to NM do
            nml := [op(nml) , add(MR[i,j]*CML[i,2,j],j= 1..NML[i])];
        od;
        return na,nb,nml;
    end:
end module:
MQRR := proc(u,m,MQ,NN,DD,p)
    local t0, t1, q, r0, r1, r, n, d , B, dq,L,x;
    if modp1(Degree(m) , p) <= MQ then return false; fi;
    x := modp1(Indeterminate(u),p);
    r1 := modp1(Rem(u,m) , p);
    t0 := modp1(Zero(x) , p);
    r0 := m;
    t1 := modp1(One(x) , p);
    B := MQ;
    while modp1(Degree(r1),p)>=0 and modp1(Degree(r0),p)>B do
        q := modp1(Quo(r0,r1,'r'),p);
        dq := modp1(Degree(q) , p);
        if dq >= B then (n,d,B) := (r1,t1,dq); fi;
        (r0,r1,t0,t1) := (r1,r,t1,modp1('Subtract'(t0,'Multiply'(q,t1)),p));
    od;
    if not assigned(n) then return false; fi;
    if modp1(Degree(Gcd(n,d)),p) > 0 then return false; fi;
    L := modp1(Lcoeff(d) , p);
    if not L = 1 then
```

```
        L := modp1(Constant(1/L mod p ,x),p);
        d := modp1(Multiply(L,d) , p);
        n := modp1(Multiply(L,n) , p);
    fi;
    NN,DD := n,d;
    true;
end:
myRatrecon := proc(U,M,x,MQ,p)
    global last_monomial;
    local y,c,tt,n,i,r,NN,DD,u,m , nn,dd;
    y := indets(U,name) minus {x};
    if nops(y) > 0 then
        c := coeffs(U,y,'tt');
        c := [c]; tt := [tt]; n := nops(c);
        if member(last_monomial,tt,'i') then else i := n fi;
        to n do
            r[i] := myRatrecon(c[i],M,x,MQ ,p);
            if r[i] = FAIL then last_monomial := tt[i]; return FAIL; fi;
            if i = 1 then i := n else i := i-1 fi;
        od;
        return add(r[i]*tt[i], i=1..n);
    fi;
    u := modp1(ConvertIn(expand(U) mod p,x),p);
    m := modp1(ConvertIn(expand(M) mod p,x),p);
    if MQRR(u,m,MQ,NN,DD,p) then
        nn := modp1(ConvertOut(NN,x),p) mod p;
        dd := modp1(ConvertOut(DD,x),p) mod p;
        return nn/dd;
    fi;
    FAIL;
end:
mSolve := module()
    option package;
    export construct_matrix , solve_system;
    local M, matrix_size, sz;
    construct_matrix := proc(gl,pl,coft,p,nt,maxu,mn,dat_t,vzl,term_length)
        local zero_counter,i,RA,gcoeffs,j,gt,NA;
```

```
        sz := mn + (nt*maxu);
        matrix_size := add(term_length[i] , i = 1..nops(term_length));
        zero_counter := 0; i := 1;
        RA := Create(p,nt,maxu,0,dat_t);
        Mod(p,coft, pl[i],RA);
        M := Matrix(sz,mn + matrix_size , datatype = dat_t);
        gcoeffs := [coeffs(expand(gl[1]) , vzl)];
        for j from 1 to nt do
            gt := gcoeffs[j];
            M[j,zero_counter+1..zero_counter+term_length[j]]:= RA[j,
                1..term_length[j]];
            M[j,-1] := gt;
            zero_counter := zero_counter + term_length[j];
        od;
        for i from 2 to mn do
            Mod(p,coft, pl[i],RA);
            zero_counter := 0;
            gcoeffs := [coeffs(expand(gl[i]) , vzl)];
            for j from 1 to nt do
                gt := gcoeffs[j];
                NA := (i-1)*nt + j;
                if (NA > sz) then break; fi;
                M[NA,zero_counter+1..zero_counter+term_length[j]]:=RA[j,
                    1..term_length[j]];
                M[NA , i - mn - 2] := gt;
                zero_counter := zero_counter + term_length[j];
            od;
        od;
    end:
    solve_system := proc(p,dat_t,mn)
        local sol;
        sol := Mod(p , M , dat_t);
        RowReduce(p,sol,sz,matrix_size+mn,matrix_size+mn,0,0,0,0,'INCROW',true);
        if INCROW <> 0 then return "Bad_Form"; fi;
        return sol[1..sz,matrix_size + mn];
    end:
end module:
```

```
nRatrecon := proc(G,mc,t,p)
    myRatrecon(G,mc,t,2,p);
end:
n_iratrecon := proc(G,mc)
    iratrecon(G,mc);
end:
PrimitiveAssociate := proc(f , s)
    local opf , den;
    den := denom(f);
    if op(0 , f) = '*' then return primpart(expand(den*f) , s); fi;
    opf := [op(f)];
    primpart(expand(add(expand(den*opf[i]),i=1..nops(opf))),s);
end:
lcbadP := proc(f1, f2, ml, p, x, zl)
    local lm, l1, l2,lml;
    lml := map(proc(mi,zi) lcoeff(mi , zi) end proc , ml , zl);
    l1 := lcoeff(f1, x);
    l2 := lcoeff(f2, x);
    for lm in lml do if lm mod p = 0 then return true; fi; od;
    if (l1 mod p = 0) or (l2 mod p = 0) then return true; fi;
    false;
end:
#MBM
Phi := proc(f,t,z,p) local d,s,i,n;
    # Compute eval(f,t=z) mod p efficiently
    if type(f,integer) then return f mod p fi;
    d := degree(f,t); s := series(f,t,d+1); n := nops(s);
    # Because 0 ^ 0 is undefined, we need to test for z=0 directly
    if z=0 then coeff(s,t,0) else
        add( op(2*i-1,s) * modp(z &^ op(2*i,s),p), i=1..n/2 ) mod p;
    fi;
end:
lcbadEP := proc(f1, f2, ml, p, alpha, t, x, zl)
    local lm , l1, l2,lml;
    lml := map(proc(mi,zi) lcoeff(mi , zi) end proc , ml , zl);
    l1 := lcoeff(f1, x);
    l2 := lcoeff(f2, x);
```

```
    for lm in lml do
        if EVAL(lm , t ,alpha,p) = 0 then return true; fi;
    od;
    if EVAL(l1 , t , alpha,p) = 0 then return true; fi;
    if EVAL(l2 , t , alpha,p) = 0 then return true; fi;
    false;
end:
MTrialDivision := proc(A, B, Ml , zl , x , Q ::  name)
    local a,b,m,n,dl,ca,cb,t1,lb,lml,r,lr,g,t2,cq,c,bl,t,i,d,lm,zz,s,q,p;
    m := degree(A , x);
    n := degree(B , x);
    dl := map (proc(mi,zi) degree(mi , zi) end proc,Ml , zl);
    ca := content(A , [x,op(zl)]);
    cb := content(B , [x,op(zl)]);
    divide(A , ca , 'a');
    divide(B , cb , 'b');
    lb := lcoeff(b , x);
    lml := map(proc(mi,zi) lcoeff(mi , zi) end proc , Ml , zl);
    r := a; s := 1; q := 0;
    while (r <> 0) and (m >= n) do
        lr := lcoeff(r , x);
        g := gcd(content(lr , zl) , lb);
        divide(lr , g , 'lr');
        divide(lb , g , 't1');
        s := expand(t1*s);
        t := expand(lr * x^ (m-n));
        divide(t , s , 't2');
        q := expand(q + t2);
        r := expand(t1*r - t*b);
        p := 1;
        for i from 1 to nops(Ml) do
            d := dl[i];
            lm := lml[i];
            zz := zl[i];
            while (r <> 0) and (degree(r , zz) >= d) do
                lr := lcoeff(r , zz);
                g := gcd(content(lr , x) , lm);
```

```
                divide(lr , g , 'lr');
                t := expand(lr*zz^ (degree(r , zz) - d));
                divide(lm , g , 't2');
                p := expand(p * t2);
                r := expand(t2*r - t*Ml[i]);
            od;
        od;
        s := s * p;
        m := degree(r , x);
    od;
    if r <> 0 then return false fi;
    bl := divide(ca , cb , 'cq');
    if not bl then return false fi;
    if nargs > 5 then Q := expand(cq*q); fi;
    return true;
end:
PTrialDivision := proc(A, B,Ml,zl, x , p)
    local a,b,m,n,d,ca,cb,t1,lb,lm,r,lr,g,t2,cq,c,bl,t,lml,dl,zz,i;
    m := degree(A , x);
    n := degree(B , x);
    dl := map (proc(mi,zi) degree(mi , zi) end proc,Ml , zl);
    a := A; b := B;
    lb := lcoeff(b , x);
    lml := map(proc(mi,zi) lcoeff(mi , zi) end proc , Ml , zl);
    r := a;
    while (r <> 0) and (m >= n) do
        lr := lcoeff(r , x);
        g := Gcd(Content(lr , zl) mod p , lb) mod p;
        if g <> 1 then
            Divide(lr , g , 'lr') mod p;
            Divide(lb , g , 't1') mod p;
        else t1 := lb;
        fi;
        t := expand(lr * x^ (m-n)) mod p;
        r := expand(t1*r - t*b) mod p;
        for i from 1 to nops(Ml) do
            d := dl[i];
```

```
            lm := lml[i];
            zz := zl[i];
            while (r <> 0) and (degree(r , zz) >= d) do
                lr := lcoeff(r , zz);
                g := Gcd(Content(lr , x) mod p, lm) mod p;
                if g <> 1 then Divide(lr , g , 'lr') mod p; fi;
                t := lr*zz^ (degree(r , zz) - d);
                if g <> 1 then Divide(lm , g , 't2') mod p; else t2 := lm; fi;
                r := expand(t2*r - t*Ml[i]) mod p;
            od;
        od;
        m := degree(r , x);
    od;
    if r <> 0 then return false; fi;
    return true;
end:
MinIndex := proc(l)
    local m;
    if nops(l) = 1 then return 1; fi;
    m := MinIndex(l[2..-1]) + 1;
    if l[m] > l[1] then return 1 else return m; fi;
end:
mContent := proc(f , x , ml , zl )
    local cl,dl,mindex , f1,bound,rgen,f2,i,r,g,k;
    if type(f , list) then
        cl := [coeffs(f[1],x),coeffs(f[2],x)];
    else
        cl := [coeffs(f , x)];
    fi;
    if nops(cl) = 1 then return cl[1]; fi;
    dl := map(degree , cl);
    mindex := MinIndex(dl);
    f1 := cl[mindex];
    bound := 10^ 2;
    rgen := rand(-bound..bound);
    f2 := 0;
    for i from 1 to nops(cl) do
```

```
        if i = mindex then next; fi;
        r := rgen();
        f2 := f2 + r*cl[i];
    od;
    f2 := expand(f2);
    g := ModGcd(f1 , f2 , ml, zl);
    if g = "failed" or g = 1 then return 1; fi;
    k := 0;
    while k < nops(cl) do
        k := k + 1;
        if k = mindex then next; fi;
        if not MTrialDivision(cl[k], g, ml , zl , x) then
            g := ModGcd(f1 , cl[k] , ml , zl);
            if g = "failed" or g = 1 then return 1; fi;
            k := 0;
        fi;
    end:
    return g;
end:
RConvert := proc(e , zl , al)
    local i , m;
    m := e;
    for i from 1 to nops(al) do m := SUBS(zl[i] = al[i] , m); od;
    m;
end:
RootConvert := proc(rl , n)
    local v,sn,ml,m,rln,vl;
    ml := [];
    m := op(SUBS(_Z=z[n] , rl[1]));
    if nops(rl) = 1 then return [m],[z[n]];
fi;
    rln := map(proc(x) SUBS(rl[1]=z[n],x) end proc , rl[2..-1]);
    rln , vl := RootConvert(rln , n + 1);
    return [op(rln),m] , [op(vl),z[n]] ;
end:
SparseInterp := proc(A , B , pattern , x, tl , zl, ml, p)
    local a,b,maxu,r,c,t,v,nt,k,i,l,temp,pl,gl,tt,na,nb,g,s,gt,res,M,sol,coft
```

```
,pat,mn,j,nml,coft_nm,coft_cnt,matrix_size,term_length,sz,vzl,dat_t
,z,u,dpatx,dax,dbx,dmlx,pat_nt,gnt,dl;
k := nops(tl);
if (k < 1) then return EUCLID(A,B,[x,op(zl)],p,ml); fi;
a , b := A , B; pat := pattern;
r := rand(1..p - 1);
v := x; l := tl;
c := [coeffs(pat, [v , op(zl)] , 't')]; t := [t];
coft_cnt := 0; maxu := 0; coft := []; coft_nm := [0];
nt := nops(t);
for i from 1 to nt - 1 do
    coeffs(c[i], l , 'temp');
    coft := [op(coft) , [temp]];
    maxu := max(maxu, nops([temp]));
    coft_cnt := coft_cnt + nops([temp]);
    coft_nm := [op(coft_nm) , coft_cnt];
od;
coeffs(c[-1], l , 'temp');
coft := [op(coft) , [temp]];
pat_nt := nops(c);
if pat_nt = 1 then
    mn := maxu + 1
else
    mn := maxu + ceil(maxu / (pat_nt - 1));
fi;
pl := []; gl := [];
vzl := [v,op(zl)];
if p < 10 ^ 6 then dat_t:=integer[4]; else dat_t:=integer[8]; fi;
EVAL_SUBS [:- initialize](a,b,vzl,zl,dat_t,ml,p);
z := 0; u := 0;
dpatx := degree(pat, x);
dax , dbx := degree(a , x) , degree(b , x);
dmlx := map(degree , ml , x);
while nops(pl) < mn do
    tt := {seq(v = r() , v = op(l)) };
    if member(tt,{op(pl)}) then next; fi;
    na,nb,nml := EVAL_SUBS [:- substitute](tt,p);
```

```
        if dax < degree(na , x) or dbx < degree(nb , x) then next; fi;
        dl := map(degree , nml , x);
        if evalb(dl <> dmlx) then next; fi;
        g := EUCLID(na , nb,vzl,p,nml);
        if g = "failed" then
            z := z + 1;
            if nops(gl) + 1 < z then return "ZeroDivisor_Prime"; fi;
            next;
        fi;
        if degree(g , x) < dpatx then
            return "Bad_Form";
        elif degree(g , x) > dpatx then
            u := u + 1;
            if nops(gl) + 1 < u then return "UNLUCKY"; fi;
            next;
        fi;
        gnt := nops([coeffs(expand(g) , [v,op(zl)])]);
        if gnt > pat_nt then
            return "Bad_Form";
        fi;
        pl := [op(pl),tt];
        gl := [op(gl),g];
    od;
    term_length := map(nops , coft);
    MATRIX_SOLVE [:-construct_matrix](gl,pl,coft,p,nt,maxu,mn,dat_t,vzl,
        term_length);
    sol:=MATRIX_SOLVE [:-solve_system](p,dat_t,mn);
    res:=add(add(coft[i][j]*sol[coft_nm[i]+j],j=1..term_length[i])*t[i],
        i=1..nt);
    if res = 0 then print("ZERO_ERROR"); return "Bad_Form"; fi;
    return res;
end:
mgcd_chrem := proc(G1,mc1,g,p)
    local In,v,G,mc;
    G,mc := G1, mc1;
    In := 1/mc mod p;
    v := In*(g-G) mod p;
```

```
    G := G + mc*v;
    mc := mc * p;
    G,mc;
end:
pgcd_chrem := proc(G1,mc1,g,alpha,t,p)
    local delta,In,v,G,mc;
    G,mc := G1, mc1;
    delta := EVAL(G , t , alpha,p);
    In := Rem(mc,t-alpha,t);
    In := 1/In mod p;
    v := In*(g-delta) mod p;
    G := G + expand( v*mc) mod p;
    mc := expand((t-alpha)*mc) mod p;
    G,mc;
end:
# Input:  f̌1, f̌2 and the list of minimal polynomials.
# Output:  ǧ, where g is the gcd of f1 and f2.
MGCD := proc(f1, f2, ml, zl, x, tl,vl)
    local pbound , pgen, p , n,d, g, G, mc, h,i,Q,tt,pat, LM, LC,GT;
    pbound := modp1(Prime(1))+1;
    p := 1; n := 1; d := 1;
    pat := 0;
    while true do
        p := modp1(Prime(p));
        while lcbadP(f1,f2,ml,p,x,zl) do p := modp1(Prime(p)); od;
            g := PGCD(f1, f2, p, ml , zl , x , tl,vl,pat);
            If g = "contfailed" then next; fi;
            If g = "ZeroDivisor_Prime" or g = "Unlucky" then next; fi;
            If g = "Bad_Form" then pat := 0; next; fi;
            if g <> "failed" then
                if g = 1 then return 1; fi;
                LC := lcoeff(g,[x,op(tl)],'LM');
                if not assigned(GT[LM]) then
                    G := expand(g);
                    mc := p;
                    GT[LM] := G,mc;
                elif degree(g,x) < degree(G,x) then
```

```
                G := g;
                mc := p;
                pat := g;
                GT[LM] := G,mc;
            elif degree(g,x) > degree(G,x) then
                next;
            else
                G,mc := MCHREM(GT[LM],g,p);
                GT[LM] := G,mc;
            fi;
            n := n + 1;
            h := I_RATRECON(G,mc);
            if h <> FAIL then
                h := PrimitiveAssociate(h,{op(zl),x});
                pat := g;
                if MDIV(f1, h, ml, zl, x) and MDIV(f2, h, ml, zl , x) then
                    return h;
                fi;
            else pat := g; fi;
            fi;
        else
            d := d + 1;
            if d > n then return "failed"; fi;
            pat := 0;
        fi;
    od;
end:
# Input:  f̌1, f̌2, p, the list of minimal polynomials and the assumed form.
# Output:  Either ǧ or an error message if the algorithm fails to compute the gcd.
PGCD := proc(f1 , f2, p, ml, zl, x, tl , vl , pat)
    local k,G,n,d,alpha,rg,t,cml,ff1,ff2,g,h,mc,c,tt,eg,i,LC,LM,ptr,D,E,F,GT;
    if CONTENT([f1,f2] , [op(1..-2,tl),op(vl)] , ml , zl) <> 1 then return "contfailed"; fi;
    k := nops(tl);
    if k = 0 then return EUCLID(f1,f2,vl,p,ml); fi;
    if (pat = 0) then
        n :=1; d :=1;
        rg := rand(1..p-1);
```

```
t := tl[k];
alpha := rg();
while lcbadEP(f1, f2, ml, p, alpha, t, x, zl) do
    alpha := rg();
od;
ff1 := EVAL(f1 , t , alpha , p);
ff2 := EVAL(f2 , t , alpha , p);
cml := map(proc(m) EVAL(m,t,alpha,p); end proc , ml);
ptr := PGCD(ff1, ff2, p, cml, zl, x, tl[1..k-1] , vl,0);
if ptr = "contfailed" then return "contfailed"; fi;
if ptr = "ZeroDivisor_Prime" or ptr = "failed" then return "ZeroDivisor_Prime"; fi;
if ptr = 1 then return 1 fi;
while true do
    alpha := rg();
    while lcbadEP(f1, f2, ml, p, alpha, t, x, zl) do alpha := rg(); od;
    cml := map(proc(m) EVAL(m,t,alpha,p); end proc , ml);
    ff1 := EVAL(f1 , t , alpha , p);
    ff2 := EVAL(f2 , t , alpha , p);
    g := SparseInterp(ff1,ff2,ptr,x,tl[1..k-1],zl,cml,p);
    if g = "Bad_Form" then
        return PGCD(f1 , f2, p, ml, zl, x, tl , vl , 0);
    fi;
    if g = "failed" then
        d := d + 1;
        if d > n then return "ZeroDivisor_Prime" fi;
    else
        LC := lcoeff(g,[x,op(tl)],'LM');
        LC := 1/LC mod p;
        g := g*LC mod p;
    fi;
    if g <> "failed" and g <> "Unlucky" then
        if g = 1 then return 1; fi;
        if not assigned(GT[LM]) then
            G := expand(g) mod p;
            mc := t - alpha;
            GT[LM] := G,mc;
        elif degree(g,x) < degree(G,x) then
```

```
                ptr := g;
                G := g;
                mc := p;
                GT[LM] := G,mc;
            elif degree(g,x) > degree(G,x) then
                next;
            else
                G,mc := PCHREM(GT[LM],g,alpha,t,p);
                GT[LM] := G,mc;
            fi;
            n := n + 1;
            h := RATRECON(G,mc,t,p);
            if h <> FAIL then
                h := PrimitiveAssociate(h,{op(zl),x});
                if PDIV(f1,h,ml,zl,x,p) and PDIV(f2,h,ml,zl,x,p) then
                    return h;
                fi;
            fi;
        fi;
    od;
else
    h := SparseInterp(f1 , f2 , pat , x, tl , zl, ml, p);
    if h = "Bad_Form" then return "Bad_Form"; fi;
    LC := lcoeff(h,[x,op(tl)],'LM');
    LC := 1/LC mod p;
    h := h*LC mod p;
    return h;
fi;
end:
ModGcd := proc(f1, f2, ml, zl)
    local ma , f1a , f2a , x , ntl, c, g, ct, G,R,tl,xl,V,vl;
    V := indets(ml);
    tl := [op(V minus {op(zl)})];
    xl := [op(indets([f1,f2]) minus V)];
    if nops(xl) = 0 then return 1; fi;
    x := xl[1];
    vl := [x , op(zl)];
```

```
    ma:=map(proc(m) PrimitiveAssociate(m ,[op(zl),x]) end,ml);
    f1a := PrimitiveAssociate(f1 , [op(zl),x]);
    f2a := PrimitiveAssociate(f2 , [op(zl),x]);
    if nops(xl) = 1 then return MGCD(f1a,f2a,ma,zl,x,tl , vl); fi;
    ntl := [op(tl) , op(2..-1,xl)];
    c := CONTENT([f1 , f2] , x , ml , zl);
    g := MGCD(f1a,f2a,ma,zl,x,ntl,vl);
    if g = "failed" then return g; fi;
    ct := CONTENT(g , x , ml , zl);
    MTrialDivision(g, ct, ma , zl , x , 'G');
    return expand(c*G);
end:
```

Bibliography

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

[2] Bruce W. Char, Keith O. Geddes, and Gaston H. Gonnet. Gcdheu: Heuristic polynomial gcd algorithm based on integer gcd computation. *J. Symb. Comput.*, 7(1):31–48, 1989.

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

[4] Jennifer de Kleine, Michael Monagan, and Allan Wittkopf. Algorithms for the non-monic case of the sparse modular gcd algorithm. In *ISSAC '05: Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, pages 124–131, New York, NY, USA, 2005. ACM Press.

[5] 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.

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

[7] Erich Kaltofen. Sparse hensel lifting. In *EUROCAL '85: Research Contributions from the European Conference on Computer Algebra-Volume 2*, pages 4–17, London, UK, 1985. Springer-Verlag.

[8] 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.

[9] Mark van Hoeij and Michael Monagan. A modular gcd algorithm over number fields presented with multiple extensions. In *ISSAC '02: Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 109–116, New York, NY, USA, 2002. ACM Press.

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

[12] Paul S. Wang. An improved multivariate polynomial factorization algorithm. *Math. Comp.*, 32(144):1215–1231, 1978.

[13] Paul S. Wang. The eez-gcd algorithm. *SIGSAM Bull.*, 14(2):50–60, 1980.

[14] Paul S. Wang. A p-adic algorithm for univariate partial fractions. In *SYMSAC '81: Proceedings of the fourth ACM symposium on Symbolic and algebraic computation*, pages 212–217, New York, NY, USA, 1981. ACM Press.

[15] Paul S. Wang, M. J. T. Guy, and J. H. Davenport. P-adic reconstruction of rational numbers. *SIGSAM Bull.*, 16(2):2–3, 1982.

[16] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *EUROSAM '79: Proceedings of the International Symposiumon on Symbolic and Algebraic Computation*, pages 216–226, London, UK, 1979. Springer-Verlag.

[17] Richard Zippel. *Effective polynomial Computation.* Kluwer Academic Publishers, 1993.