A New Black Box GCD Algorithm Using Hensel Lifting

Michael Monagan[1][0000−0002−4652−2889] and Garrett Paluck[1][0000−0001−9498−1011]

Simon Fraser University, Burnaby, BC V5A 1S6, Canada

Abstract. We present a new black box GCD algorithm for two multivariate polynomials a and b in $\mathbb{Z}[x_1, x_2, \ldots, x_n]$ where a and b are input as black boxes for their evaluation. Our algorithm computes $g = \gcd(a, b)$ in the sparse representation using sparse Hensel lifting from bivariate images of g. More precisely, our algorithm first computes the square-free factorization of the primitive part of g in x_1 and then, optionally, computes the content of g in x_1 recursively. We have implemented our new algorithm in Maple with parts of it coded in C for increased efficiency. For comparison, we have implemented the Kaltofen–Diaz black box GCD algorithm and also a black box GCD algorithm constructed from the Kaltofen–Yang sparse rational function interpolation algorithm. Our experimental results show that our new algorithm is always competitive with the Kaltofen–Yang and Kaltofen–Diaz algorithms and faster when the square-free factors of g are smaller than g or we do not need the content of g in x_1.

Keywords: black box representation · multivariate polynomial GCD · sparse Hensel lifting

1 Introduction

Let a and b be polynomials in $\mathbb{Z}[x_1, ..., x_n]$. Computing $g = \gcd(a, b)$, the greatest common divisor (GCD) of a and b, is a key operation in a Computer Algebra system. It is used to simplify the rational function a/b. The first main step to factor a is to compute $\gcd(a, \partial a/\partial x_1)$ to identify repeated factors of a.

Computing GCDs in $\mathbb{Z}[x_1, \ldots, x_n]$ is more difficult than multiplying and dividing polynomials. All variations of the Euclidean algorithm, including the Subresultant algorithm (see Brown and Traub [2]), encounter an n dimensional expression swell where the intermediate remainders grow in size. This renders those algorithms useless when n is not small. The first algorithm to avoid the expression swell was Brown's dense modular GCD algorithm from [3]. Two early sparse GCD algorithms for sparse polynomials include Zippel's sparse GCD algorithm from [23] which is currently used in Fermat, Magma, Maple and Mathematica, and Wang's EEZ-GCD algorithm from [21]. Two recent sparse GCD algorithms include Hu and Monagan [11] which does a Kronecker substitution on $(x_2, ..., x_n)$ and Huang and Monagan [12] which uses a randomized Kronecker substitution.

In this work, we present a new GCD algorithm for $\mathbb{Z}[x_1, \ldots, x_n]$ where the polynomials a and b are input by black boxes for their evaluation. The black box representation was first introduced into Computer Algebra by Kaltofen and Trager in 1990 [13]. The *sparse representation* of $a \in \mathbb{Z}[x_1, ..., x_n]$ consists of a list of non-zero integer coefficients c_k and monomials M_k in $x_1, ..., x_n$ such that $a = \sum_{k=1}^{t} c_k M_k$ where t is the number of terms of a. The *black box representation* of $a \in \mathbb{Z}[x_1, ..., x_n]$ is a computer program $\mathbf{B}_a$ that takes a point $\alpha \in \mathbb{Z}^n$ and outputs $a(\alpha)$. Computing $\mathbf{B}_a(\alpha)$ is called *probing the black box*. For efficiency, we will assume we can construct a *modular black box*, that is, a black box that can compute $a(\alpha) \bmod p$ for a prime p. Thus we view the black box as mapping $\mathbf{B}_a : (\mathbb{Z}^n, p) \to \mathbb{Z}_p$. Figure 1 depicts a modular black box.

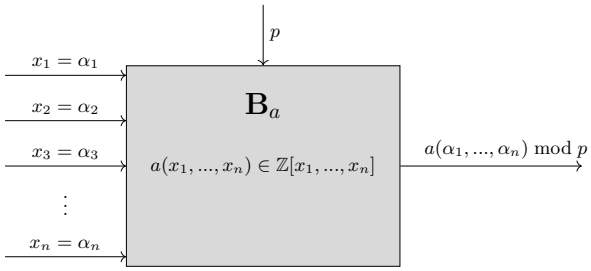

Fig. 1. The modular black box model for $a \in \mathbb{Z}[x_1, ..., x_n]$

The advantage of the black box representation is that computing $\mathbf{B}_a(\alpha, p)$ can be much faster than computing $a(\alpha) \bmod p$ in the sparse representation. For example, the black box representation of the polynomial $a = (x_1 - x_2)(x_1 - x_3)\cdots(x_1 - x_n)$ can be represented using $O(n)$ space and it can be evaluated using just $n - 1$ subtractions and $n - 2$ multiplications. But, in the sparse representation, a has 2^{n-1} terms which is exponential in n.

In [13], Kaltofen and Trager presented algorithms for factoring polynomials and computing the GCD of polynomials given by black boxes. In [7], Kaltofen and Diaz improved the black box GCD algorithm of Kaltofen and Trager [13]. The only other black box GCD algorithm that we know of is Lecerf and van der Hoeven's algorithm in [19] which uses ideas similar to the sparse rational function interpolation of Cuyt and Lee [6]. The recent Hu–Monagan [11] and Huang–Monagan [12] GCD algorithms are not black box algorithms and cannot easily be made into black box algorithms.

In [13], Kaltofen and Trager also presented an algorithm that, given a rational function $f = a/b$, outputs black boxes for polynomials $c = a/g/u$ and $d = b/g/u$ for some unit u. Kaltofen and Trager called this operation the separation of numerators from their denominators. In [14], Kaltofen and Yang improved on Kaltofen and Trager's algorithm. One way to compute $g = \gcd(a, b)$ is to first compute c and d then use $g = a/c$ or $g = b/d$ to obtain g. We will compare this approach using Kaltofen–Yang with our new algorithm. Kaltofen and Yang, Kaltofen and Diaz, and van der Hoeven all reduce the problem of computing black box GCDs to polynomial interpolation. We pair these algorithms with the Ben-Or/Tiwari sparse interpolation algorithm [1]. We do not use Giorgi et. al. [10] which is less efficient in our multivariate context.

Our new algorithm uses the Hensel lifting algorithm from Chen and Monagan [5]. It first computes $\text{pp}(g, x_1)$, the primitive part of $g = \gcd(a, b)$ in x_1, using Hensel lifting. It recovers the variables x_j for $j = 2, 3, \ldots, n$ in $\text{pp}(g, x_1)$ from bivariate images of $\text{pp}(g, x_1)$ in x_1 and x_j which are obtained by probing the black boxes for a and b. Bivariate images are used to recover the leading coefficient of $\text{pp}(g, x_1)$. Then, in a second step, it computes $\text{cont}(g, x_1)$, the content of g in x_1 recursively. An advantage of our approach is that we can easily omit computing $\text{cont}(g, x_1)$. Thus, our algorithm will be faster than Kaltofen–Diaz when $\text{pp}(g, x_1)$ is smaller than g. One application where the content is not needed is when we want to solve $\{a(x_1, x_2, \ldots, x_n) = 0, b(x_1, x_2, \ldots, x_n) = 0\}$ for x_1. Our algorithm outputs the GCD in the sparse representation as in van der Hoeven–Lecerf [20].

To further improve efficiency, we recover the square-free factorization of $\text{pp}(g, x_1)$. We can do this without increasing the overall asymptotic cost by first computing the square-free part of the bivariate images of g in x_1 and x_j then using the bivariate Hensel lifting algorithm from [17] to recover a square-free factorization in x_1 and x_j. This gives our algorithm a second advantage; it will be faster whenever the square-free factors of $\text{pp}(g, x_1)$ are smaller than $\text{pp}(g, x_1)$.

All previous black box GCD algorithms work over fields of large cardinality. To compute monic(g) over $\mathbb{Q}$, for efficiency, we use a modular GCD algorithm, that is, we compute g modulo primes and recover the rational coefficients of monic(g) using Chinese remaindering and rational number reconstruction (see [22,15]). If we don't do this, there is a large efficiency loss caused by arithmetic with large intermediate rationals in the black-box GCD algorithms.

In the modular GCD algorithm, the primes used must not divide the integer leading coefficient of g. In the sparse representation, Brown [3] imposes this requirement by requiring that a prime p not divide the leading coefficient of the input polynomial a. In the black box model, we do not have access to the leading coefficient of a, so we cannot enforce this requirement a priori, which means that our computed GCD modulo p may not be a multiple of g modulo p. In our implementation we will, eventually, discard such bad images.

We have implemented our GCD algorithm in Maple with some subalgorithms coded in C for efficiency. Our code is available for download at http://www.cecm.sfu.ca/~mmonagan/code/BBMGCD/ . It uses 62 bit primes, which are the largest machine primes that can be used in Maple.

Our paper is organized as follows. Our new algorithm for computing the GCD of polynomials a and b in $\mathbb{Z}[x_1, ..., x_n]$ modulo a prime p where a and b are given as black boxes for their evaluation is presented in Section 3. We present also an improvement to the CMSHL algorithm created by Chen and Monagan in [4]. Section 4 gives a complexity analysis for our algorithm. Section 5 presents three benchmarks that compare our new algorithm with the two other approaches mentioned above. Section 6 presents implementation details for some of the subalgorithms that we use.

2 Definitions and Notation

We fix the lexicographical monomial ordering of polynomials in this paper with $x_1 > \cdots > x_n$. For a polynomial $a \in \mathbb{Z}[x_1, ..., x_n]$, we denote $\mathrm{LC}(a)$ as the leading coefficient of a and $\mathrm{LM}(a)$ as the leading monomial. We say a is monic if $\mathrm{LC}(a) = 1$ and define $\mathrm{monic}(a) = a/\mathrm{LC}(a)$. We denote the number of terms in the polynomial a by $\#a$.

Definition 1. *Let $a, b \in \mathbb{Z}[x_1, ..., x_n]$. Then $g = \gcd(a, b)$ is the greatest common divisor (GCD) of a and b if (i) g divides a and b, (ii) every common divisor of a and b divides g, and (iii)* $\mathrm{sign}(\mathrm{LC}(g)) = +1$. *Note that (iii) imposes uniqueness on g.*

Definition 2. *Let $a \in \mathbb{Z}[x_2, ..., x_n][x_1]$ with degree $d = \deg(a, x_1)$. If $a = \sum_{i=0}^{d} a_i x_1^i$, we define $\mathrm{LC}(a, x_1) = a_d$, a polynomial in $\mathbb{Z}[x_2, ..., x_n]$. We define the content of a in x_1 by $\mathrm{cont}(a, x_1) = \gcd(a_0, a_1, ..., a_d)$, a polynomial in $\mathbb{Z}[x_2, ..., x_n]$. If $\mathrm{cont}(a, x_1) = 1$, we say a is primitive in x_1. We define the primitive part of a in x_1 by $\mathrm{pp}(a, x_1) = a/\mathrm{cont}(a, x_1)$.*

Definition 3 (Definition 8.1 in [9]). *Let $a \in \mathbb{Z}_p[x_1, ..., x_n]$ be primitive in x_1. We say a is square-free if it has no repeated factors, that is, there exists no b with $\deg(b) \geq 1$ such that $b^2 | a$. The square-free factorization of a is $a = \prod_{i=1}^{r} a_i^i$ where each a_i is square-free and $\gcd(a_i, a_j) = 1$ for $i \neq j$. The square-free part of a, denoted sqf(a), is defined as $\mathrm{sqf}(a) = \prod_{i=1}^{r} a_i$.*

Lemma 1. *Let $a \in \mathbb{Z}[x_1, ..., x_n]$, $a \neq 0$ and $g = \gcd(a, \partial a/\partial x_1)$. Then $a/g = \mathrm{sqf}(\mathrm{pp}(a, x_1))$ since $\mathrm{cont}(a, x_1) | g$.*

Example 1. Given $g = 3(x_2 - x_3)(x_1 - x_2)^2(x_1 + x_3)$, we have $\mathrm{cont}(g, x_1) = 3(x_2 - x_3)$ and $\mathrm{pp}(g, x_1) = (x_1 - x_2)^2(x_1 + x_3)$. The square-free factorization of $\mathrm{pp}(g, x_1)$ is $(x_1 + x_3)^1(x_1 - x_2)^2$ and $\mathrm{sqf}(\mathrm{pp}(g, x_1)) = (x_1 - x_2)(x_1 + x_3)$.

Our black box GCD algorithm will first compute the square-free factorization $(x_1-x_2)^2(x_1+x_3)$ of $\text{pp}(g, x_1)$ in factored form. It will then compute $\text{cont}(a, x_1)/3 = (x_2 - x_3)$. Here all factors have only two terms. This is faster than computing $\text{monic}(g) = (x_2-x_3)(x_1-x_2)^2(x_1-x_3)$ in expanded form which has 10 terms.

3 Our New Black Box GCD Algorithm

Let a and b be polynomials in $\mathbb{Z}[x_1, ..., x_n]$ with associated modular black boxes $\mathbf{B}_a$ and $\mathbf{B}_b$, and let $g = \gcd(a, b)$. In this section, we present two new algorithms. Algorithm 2: MHL_BB_PGCD computes $\text{monic}(g)$ mod p in $\mathbb{Z}_p[x_1, ..., x_n]$ for a prime p and Algorithm 3: CM_BB_SHL_GCD lifts $\text{monic}(\text{sqf}(\gcd(a(x_1, \alpha), b(x_1, \alpha))))$ mod p to $\text{monic}(\text{sqf}(\gcd(a, b)))$ mod p for an evaluation point $\alpha \in \mathbb{Z}_p^{n-1}$ using sparse Hensel lifting. We combine Algorithm 2 with Chinese remaindering and rational number reconstruction (see [22,15]) to compute $\text{monic}(g) \in \mathbb{Q}[x_1, ..., x_n]$ from the modular images. We shall refer to the algorithm that computes $\text{monic}(g)$ as the BB_MGCD algorithm. As it's a conceptually simple algorithm, we will not provide pseudocode for it. The BB_MGCD algorithm is similar in concept to Algorithm 7.1 in [9].

Computing the degrees of a, b and g Algorithm 2 requires the degree estimates $\deg(a, x_i)$, $\deg(b, x_i)$, and $\deg(g, x_i)$ for $1 \le i \le n$. To compute $\deg(a, x_i)$, we pick $\alpha \in \mathbb{Z}_p^n$ at random then apply Algorithm 1 from [4] to interpolate $a_i = a(\alpha_1, ..., \alpha_{i-1}, x_i, \alpha_{i+1}, ..., \alpha_n)$ mod p. We have $\deg(a_i, x_i) = \deg(a, x_i)$ with high probability (see [4,7]). It is possible that $\deg(a_i, x_i) < \deg(a, x_i)$. This is unavoidable in the black box model.

Algorithm 1: UnivInterp: Interpolate $a(x_i)$ mod p

Input: A modular black box $\mathbf{B}_a$ for $a \in \mathbb{Z}[x_1, ..., x_n]$, $\alpha \in \mathbb{Z}_p^n$, $i \in \mathbb{N}$, and a large prime p.
Output: A polynomial $b = a(\alpha_1, ..., \alpha_{i-1}, x_i, \alpha_{i+1}, ..., \alpha_n)$ mod p s.t. $\deg(b, x_i) = \deg(a, x_i)$ w.h.p.
Reference: Kaltofen and Diaz [7], Chen and Monagan [4].
$b \leftarrow 0$. $M \leftarrow 1$. $k \leftarrow -1$.
repeat
  $k \leftarrow k + 1$.
  Pick $\beta_k \in \mathbb{Z}_p^*$ at random s.t. $\beta_k \neq \beta_j$ for $0 \le j \le k-1$.
  $y_k \leftarrow \mathbf{B}_a((\alpha_1, \alpha_2, ..., \alpha_{i-1}, \beta_k, \alpha_{i+1}, ..., \alpha_n), p)$.
  $v_k \leftarrow (y_k - b(\beta_k))/M(\beta_k)$.
  $b \leftarrow b + v_k \cdot M$.
  $M \leftarrow M \cdot (x_i - \beta_k)$.
until $v_k = 0$;
return b.

3.1 The MHL_BB_PGCD Algorithm

Consider the following square-free factorization

$$g = \gcd(a, b) = h \prod_{\rho=1}^{r} f_\rho^\rho$$

where (i) $h = \text{cont}(g, x_1)$ is in $\mathbb{Z}[x_2, ..., x_n]$, (ii) $\deg(f_\rho, x_1) \geq 0$, (iii) f_ρ is primitive and square-free in $\mathbb{Z}[x_2, ..., x_n][x_1]$, and (iv) $\gcd(f_i, f_j) = 1$ for $i \neq j$. We shall now describe our new algorithm which computes $\text{monic}(f_\rho)$ modulo a prime p for $1 \leq \rho \leq r$.

We assume that p is a large prime (e.g. $p = 2^{61} + 15$) chosen randomly in advance and that we have degree estimates da_i, db_i and dg_i for $\deg(a, x_i)$, $\deg(b, x_i)$, and $\deg(g, x_i)$ for $1 \leq i \leq n$.

We first pick an evaluation point $\alpha = (\alpha_2, ..., \alpha_n) \in \mathbb{Z}_p^{n-1}$ at random, then interpolate $a_1 = a(x_1, \alpha) \bmod p$ and $b_1 = b(x_1, \alpha) \bmod p$ in $\mathbb{Z}_p[x_1]$ using dense interpolation and probes to the black boxes $\mathbf{B}_a$ and $\mathbf{B}_b$ such that $\deg(a_1, x_1) = da_1$ and $\deg(b_1, x_1) = db_1$. We use at least $da_1 + 2$ probes when interpolating a_1 to verify that our degree estimate da_1 is correct. We do likewise for b_1. In fact, whenever we do an interpolation in this section, we use one extra probe than necessary to check our degree estimates. If they do not agree, we stop the algorithm and output FAIL.

Next we compute $g_1 = \gcd(a_1, b_1)$ in $\mathbb{Z}_p[x_1]$, check that $\deg(g_1) = dg_1$, and compute the square-free factorization of g_1. Let $g_1 = \prod_{\rho=1}^{r} \hat{f}_\rho^\rho$ be the square-free factorization of g_1 where $\hat{f}_\rho = (1/\lambda_\rho) f_\rho(x_1, \alpha) \bmod p$ and $\lambda_\rho = \text{LC}(f_\rho(x_1, \alpha)) \in \mathbb{Z}$ for $1 \leq \rho \leq r$. Let $\hat{h} = \lambda_h h(\alpha) \bmod p$ with $\lambda_h = \prod_{\rho=1}^{r} \lambda^\rho \in \mathbb{Z}$. To be explicit,

$$\begin{aligned} g(x_1, \alpha) &= h(\alpha) f_1(x_1, \alpha)^1 \cdots f_r(x_1, \alpha)^r \\ &= h(\alpha) \left(\lambda_1 \hat{f}_1\right)^1 \cdots \left(\lambda_r \hat{f}_r\right)^r \text{ w.h.p.} \\ &= \underbrace{h(\alpha) \left(\prod_{\rho=1}^{r} \lambda_\rho^\rho\right)}_{\hat{h}} \underbrace{\hat{f}_1^1 \cdots \hat{f}_r^r}_{g_1}. \end{aligned}$$

The coefficients λ_ρ can be recovered later after Chinese remaindering and the content h can be recovered recursively. For most choices of α and p, we will have (i) $\deg(f_\rho(x_1, \alpha), x_1) = \deg(\hat{f}_\rho, x_1)$, (ii) $f_\rho = \text{monic}(f_\rho(x_1, \alpha))$ for $1 \leq \rho \leq r$ and (iii) $\gcd(\hat{f}_i, \hat{f}_j) = 1$ for all $i \neq j$ which is needed for Hensel lifting.

Let $g_j = \text{monic}(\text{pp}(g(x_1, ..., x_j, \alpha_{j+1}, ..., \alpha_n), x_1)) \mod p$ and let $\hat{f}_{\rho,1} = \hat{f}_\rho$. Let $\hat{f}_{\rho,j} = \text{monic}(f_\rho(x_1, ..., x_j, \alpha_{j+1}, ..., \alpha_n))$ for $2 \leq j \leq n$ which we will compute sequentially. We call the CM_BB_SHL_GCD algorithm (to be described shortly) with inputs $\mathbf{B}_a, \mathbf{B}_b, \alpha, p, (\hat{f}_{\rho,1}, ..., \hat{f}_{\rho,r})$, da, db, and dg to Hensel lift $\hat{f}_{\rho,1}(x_1)$ to $\hat{f}_{\rho,2}(x_1, x_2)$, then lift $\hat{f}_{\rho,2}(x_1, x_2)$ to $\hat{f}_{\rho,3}(x_1, x_2, x_3)$, etc. After the j^{th} Hensel lifting step we've computed $\hat{f}_{\rho,j}$ s.t. $\text{sqf}(g_j) = \prod_{\rho=1}^{r} \hat{f}_{\rho,j} \bmod p$ and $\text{monic}(\hat{f}_{\rho,j}(x_j = \alpha_j)) = \hat{f}_{\rho,j-1} \bmod p$. When the CM_BB_SHL_GCD algorithm terminates, it returns either $\hat{f}_{\rho,n}$ such that $\text{sqf}(g_n) = \prod_{\rho=1}^{r} \hat{f}_{\rho,n} \pmod p$ or FAIL if it gets unlucky in its choice of evaluation points. If this happens, the CM_BB_SHL_GCD algorithm must restart with a different α and a new prime p. If we do not want h, we can stop here and return $\hat{f} = \prod_{\rho=1}^{r} \hat{f}_{\rho,n}^\rho$.

To recover $\text{monic}(\text{cont}(g, x_1))$, we construct two new modular black boxes $\mathbf{B}_c, \mathbf{B}_d : (\mathbb{Z}^{n-1}, p) \to \mathbb{Z}_p$ such that for $\beta \in \mathbb{Z}_p$ and $\gamma \in \mathbb{Z}_p^{n-1}$, $\mathbf{B}_c(\gamma, p)$ computes $a(\beta, \gamma)/\hat{f}(\beta, \gamma) \bmod p$ and $\mathbf{B}_d(\gamma, p)$ computes $b(\beta, \gamma)/\hat{f}(\beta, \gamma) \bmod p$. The black boxes return FAIL if $\hat{f}$ evaluates to 0 in which case we need to restart with a different γ. We use our MHL_BB_PGCD algorithm to get the GCD of $\mathbf{B}_c$ and $\mathbf{B}_d$ over $\mathbb{Z}_p[x_2, ..., x_n]$ recursively.

We present the MHL_BB_PGCD algorithm as Algorithm 2.

Algorithm 2: MHL_BB_PGCD - Computes monic$(\gcd(a,b))$ mod p for black boxes

Input: Modular black boxes $\mathbf{B}_a$ and $\mathbf{B}_b$ for $a, b \in \mathbb{Z}[x_1, ..., x_n]$, $X = [x_1, ..., x_n]$, $n \in \mathbb{N}$, prime p, degree estimates $da_i \le \deg(a, x_i)$, $db_i \le \deg(b, x_i)$, and $dg_i \ge \deg(\gcd(a,b), x_i)$ $(1 \le i \le n)$.

Output: $g \in \mathbb{Z}_p[x_1, x_2, ..., x_n]$ s.t. $g = \text{monic}(\gcd(a,b)) \bmod p$ or FAIL.

Pick $\alpha = (\alpha_2, ..., \alpha_n) \in (\mathbb{Z}_p \setminus \{0\})^{n-1}$ at random.
Interpolate $a_1 = a(x_1, \alpha) \in \mathbb{Z}_p[x_1]$ via $da_1 + 2$ probes to $\mathbf{B}_a$.
if $\deg(a_1, x_1) \ne da_1$ **then return** FAIL **end**
Interpolate $b_1 = b(x_1, \alpha) \in \mathbb{Z}_p[x_1]$ via $db_1 + 2$ probes to $\mathbf{B}_b$.
if $\deg(b_1, x_1) \ne db_1$ **then return** FAIL **end**
$g_1 \leftarrow \gcd(a_1, b_1) \in \mathbb{Z}_p[x_1]$. //$g_1$ is monic
if $\deg(g_1, x_1) \ne dg_1$ **then return** FAIL **end**
Find the square-free factorization $\prod_{\rho=1}^{r} \hat{f}_{\rho,1}^{\rho}$ of g_1 in $\mathbb{Z}_p[x_1]$.
if $n = 1$ **then return** $\prod_{\rho=1}^{r} \hat{f}_{\rho,1}^{\rho}$ **end**
//Calculate the primitive part of $\gcd(a \bmod p, b \bmod p)$
$[\hat{f}_{1,n}, ..., \hat{f}_{r,n}] \leftarrow$ CM_BB_SHL_GCD$(\mathbf{B}_a, \mathbf{B}_b, n, [\hat{f}_{1,1}, ..., \hat{f}_{r,1}], \alpha, p, da, db, dg)$.
if CM_BB_SHL_GCD returned FAIL **then return** FAIL **end**
$\hat{f} \leftarrow \prod_{\rho=1}^{r} \hat{f}_{\rho,n}{}^{\rho} \in \mathbb{Z}_p[x_1, ..., x_n]$. //$\hat{f} = \text{monic}(\text{pp}(\gcd(a,b), x_1)) \bmod p$
if the content of $\gcd(a,b)$ in x_1 is not needed **return** $\hat{f}$ **end**
if $dg_i - \deg(\hat{f}, x_i) = 0$ for $2 \le i \le n$ **then return** $\hat{f}$ (there is no content) **end**
// Calculate monic$(\text{cont}(g, x_1)) \bmod p$
Pick $\beta \in \mathbb{Z}_p \setminus \{0\}$ at random. //fix x_1
Create a modular black box $\mathbf{B}_c : (\mathbb{Z}^{n-1}, p) \to \mathbb{Z}_p$ which for input $\gamma \in \mathbb{Z}_p^{n-1}$ computes $\mathbf{B}_a((\beta, \gamma), p) / \hat{f}(\beta, \gamma) \bmod p$ if $\hat{f}(\beta, \gamma) \ne 0$ and FAIL otherwise.
Create a modular black box $\mathbf{B}_d : (\mathbb{Z}^{n-1}, p) \to \mathbb{Z}_p$ which for input $\gamma \in \mathbb{Z}_p^{n-1}$ computes $\mathbf{B}_b((\beta, \gamma), p) / \hat{f}(\beta, \gamma) \bmod p$ if $\hat{f}(\beta, \gamma) \ne 0$ and FAIL otherwise.
for i from 2 to n **do**
    $DA_i \leftarrow da_i - \deg(\hat{f}, x_i)$.
    $DB_i \leftarrow db_i - \deg(\hat{f}, x_i)$.
    $DG_i \leftarrow dg_i - \deg(\hat{f}, x_i)$.
end
$h \leftarrow$MHL_BB_PGCD$(\mathbf{B}_c, \mathbf{B}_d, [x_2, ..., x_n], n-1, p, DA, DB, DG)$.
if MHL_BB_PGCD returned FAIL **then return** FAIL **end**
Set $g = h \cdot \hat{f}$.
return g.

Example 2. Consider the polynomials

$$a = 6(7x_2 - 3x_3)(2x_1 + 4x_2 + 1)(x_1 - x_3)^3(x_1^2 + x_2 + x_3 + 1),$$
$$b = 4(7x_2 - 3x_3)(2x_1 + 4x_2 + 1)(x_1 - x_3)^3(x_1 + x_2^2 + x_3 + 1)$$

in $\mathbb{Z}[x_1, x_2, x_3]$ and let $\mathbf{B}_a$ and $\mathbf{B}_b$ be the modular black box representations of a and b respectively. We have $g = \gcd(a, b) = 2(7x_2 - 3x_3)(2x_1 + 4x_2 + 1)(x_1 - x_3)^3$, $\mathrm{pp}(g, x_1) = (2x_1 + 4x_2 + 1)(x_1 - x_3)^3$, $\mathrm{cont}(g, x_1) = 2(7x_2 - 3x_3)$, and

$$\mathrm{monic}(g) = (x_2 - \tfrac{3}{7}x_3)(x_1 + 2x_2 + \tfrac{1}{2})(x_1 - x_3)^3.$$

We demonstrate how algorithm MHL_BB_PGCD computes $g_m = \mathrm{monic}(g) \bmod p$ for $p = 31$. MHL_BB_PGCD is given degree estimates for a, b and g that are correct with high probability. In this case, the degrees are $da = (6, 3, 5), db = (5, 4, 5)$ and $dg = (4, 2, 4)$ for a, b and g respectively.

Let $\alpha = (\alpha_2, \alpha_3) = (7, 13)$. The MHL_BB_PGCD algorithm begins by using dense interpolation and probes to the $\mathbf{B}_a$ and $\mathbf{B}_b$ to recover $a_1 = a(x_1, \alpha_2, \alpha_3) \bmod p = 23x_1^6 + 10x_1^5 + 21x_1^4 + 18x_1^3 + 2x_1^2 + 29x_1 + 21$, and $b_1 = b(x_1, \alpha_2, \alpha_3) \bmod p = 19x_1^5 + 3x_1^4 + 4x_1^3 + 11x_1^2 + 8x_1 + 17$. After dense interpolation, we calculate

$$g_1 = \gcd(a_1, b_1) = x_1^4 + 22x_1^3 + 19x_1^2 + 24x_1 + 27.$$

Next we compute the square-free factorization of g_1 and obtain

$$g_1 = (x_1 + 30)(x_1 + 18)^3. \tag{1}$$

Next we use the CM_BB_SHL_GCD algorithm to lift the factors of $\mathrm{sqf}(g_1)$, namely $x_1 + 30$ and $x_1 + 18$, to get

$$\mathrm{sqf}(\mathrm{pp}(g_m, x_1)) = (x_1 + 2x_2 + 16)(x_1 + 30x_3) \in \mathbb{Z}_p[x_2, x_3][x_1].$$

We include the multiplicities calculated in (1) to get

$$\mathrm{pp}(g_m, x_1) = (x_1 + 2x_2 + 16)(x_1 + 30x_3)^3.$$

Next MHL_BB_PGCD makes a recursive call to calculate the GCD of the polynomial contents of a and b over $\mathbb{Z}_p$. This will return $\mathrm{cont}(g_m, x_1) = (x_2 + 4x_3)$. MHL_BB_PGCD concludes by returning $g_m = (x_2 + 4x_3)(x_1 + 2x_2 + 16)(x_1 + 30x_3)^3$. We note the factors of g_m are monic in lex order with $x_1 > x_2 > x_3$.

$\Diamond$

The CM_BB_SHL_GCD Algorithm The CM_BB_SHL_GCD algorithm has the following input and output:

Input: Modular black boxes $\mathbf{B}_a, \mathbf{B}_b : (\mathbb{Z}^n, p) \to \mathbb{Z}_p$, $\hat{f}_{\rho,1} \in \mathbb{Z}_p[x_1] (1 \le \rho \le r), \alpha \in \mathbb{Z}_p^{n-1}$, a prime p, degree estimates $da_i \le \deg(a, x_i)$, $db_i \le \deg(b, x_i)$, and $dg_i \ge \deg(g, x_i) (1 \le i \le n)$, $X = [x_1, ..., x_n]$, and $n \in \mathbb{N}$ s.t.

(i) $\gcd(\hat{f}_{k,1}, \hat{f}_{l,1}) = 1$ for $k \ne l$ in $\mathbb{Z}_p[x_1]$,

(ii) $\text{sqf}(g_1) = \prod_{\rho=1}^{r} \hat{f}_{\rho,1} \bmod p \in \mathbb{Z}_p[x_1]$.
(iii) $\hat{f}_{\rho,1}$ is monic in x_1 for all $1 \le \rho \le r$.

Output FAIL or $\hat{f}_{\rho,n} \in \mathbb{Z}_p[x_1, ..., x_n] (1 \le \rho \le n)$ s.t.

(i) $\text{sqf}(g_n) = \prod_{\rho=1}^{r} \hat{f}_{\rho,n} \bmod p \in \mathbb{Z}_p[x_1, ..., x_n]$,
(ii) $\text{monic}(\hat{f}_{\rho,n}(x_1, \alpha)) = \hat{f}_{\rho,1} \bmod p$ for all $1 \le \rho \le r$,
(iii) $\hat{f}_{\rho,n}$ is monic in lex $x_1 > x_2 > \cdots > x_n$ for $1 \le \rho \le r$.

We have modified the CMBBSHL algorithm created by Chen and Monagan in 2024 [4]. Our algorithm lifts the monic square-free factors $\hat{f}_{\rho,1}$ of g_1 to get the monic square-free factors of $\text{pp}(g, x_1)$ mod p. It lifts $\hat{f}_{\rho,1}(x_1)$ to $\hat{f}_{\rho,2}(x_1, x_2)$, then lifts $\hat{f}_{\rho,2}(x_1, x_2)$ to $\hat{f}_{\rho,3}(x_1, x_2, x_3)$, etc. After the j^{th} Hensel lifting step, $\text{sqf}(g_j) = \prod_{\rho=1}^{r} \hat{f}_{\rho,j} \bmod p$ and $\text{monic}(\hat{f}_{\rho,j}(x_j = \alpha_j)) = \hat{f}_{\rho,j-1} \bmod p$. After the n^{th} step, $\text{sqf}(g_n) = \prod_{\rho=1}^{r} \hat{f}_{\rho,n} \bmod p$.

CMBBSHL assumes $f_\rho(x_1, ..., x_j, \alpha_{j+1}, ..., \alpha_n)$ and $f_\rho(x_1, ..., x_{j-1}, \alpha_j, ..., \alpha_n)$ have the same supports in $x_1, ..., x_{j-1}$ for $2 \le j \le n$. This is true with high probability if p is large and α_i is chosen at random from $\mathbb{Z}_p$ (see [5]). Our CM_BB_SHL_GCD assumes likewise.

We present the CM_BB_SHL_GCD algorithm as Algorithm 3 and the direct sub-algorithm CM_BB_SHL_GCD_stepj as Algorithm 4.

To reduce the number of black box probes that algorithm CMBBSHL of Chen and Monagan does, we make the following change. Chen and Monagan interpolate the bivariate images

$$A_k = a(x_1, Y_k, x_j, \alpha_{j+1}, \ldots, \alpha_n) \quad \text{and} \quad B_k = b(x_1, Y_k, x_j, \alpha_{j+1}, \ldots, \alpha_n)$$

in $\mathbb{Z}_p[x_1, x_j]$ then compute their gcd $G_k = \gcd(A_k, B_k)$ then compute $S_k = \gcd(G_k, \partial G_k/\partial x_1)$ then compute the square-free part G_{sf} of G_k using $G_{sf} = G_k/S_k$. These are all bivariate computations in $\mathbb{Z}_p[x_1, x_j]$. Instead, we use rational function interpolation in steps 25 and 26 to interpolate x_j in $\text{monic}(G_{sf}) \in \mathbb{Z}_p(x_j)[x_1]$ from unvariate images in $\mathbb{Z}_p[x_1]$ computed in steps 16, 18, and 19. Then, we clear fractions in steps 27 and 28 to obtain $G_{sf} \in \mathbb{Z}_p[x_1, x_j]$. This avoids interpolating x_j in A_k and B_k. It is faster whenever $\deg(A_k, x_j)$ and $\deg(B_k, x_j)$ are greater than $\deg(G_{sf}, x_j)$.

Algorithm 3: CM_BB_SHL_GCD

Input: Modular black boxes $\mathbf{B}_a$, $\mathbf{B}_b$ for $a, b \in \mathbb{Z}[x_1, ..., x_n]$, $n \in \mathbb{Z}$, $\hat{f}_{\rho,1} \in \mathbb{Z}_p[x_1] (1 \le \rho \le r)$ s.t. conditions (i)-(iii) of the input are satisfied, $\alpha \in \mathbb{Z}_p^{n-1}$, a prime p, degree estimates da_i, db_i and dg_i for $\deg(a, x_i)$, $\deg(b, x_i)$, and $\deg(g, x_i)$ $(1 \le i \le n)$.

Output: FAIL or $\hat{f}_{\rho,n} \in \mathbb{Z}_p[x_1, ..., x_n] (1 \le \rho \le r)$ s.t. conditions (i)-(iii) of the output are satisfied.

```
1 for j from 2 to n do
2 |  [f̂_{1,j}, ..., f̂_{r,j}] ← CM_BB_SHL_GCD_stepj(B_a, B_b, [f̂_{1,j-1}, ..., f̂_{r,j-1}], α, p, da, db, dg, j).
3 |  if CM_BB_SHL_GCD_stepj returned FAIL then return FAIL end
4 end
5 return [f̂_{1,n}, ..., f̂_{r,n}]
```

In step 35 of Algorithm 4, BivariateHenselLift performs a bivariate Hensel lift. Chen and Monagan improved our monic bivariate Hensel lifting algorithm from [17] to treat the non-monic case (see Algorithm 14 in [4]).

Algorithm 4: CM_BB_SHL_GCD_stepj: Hensel lift x_j

Input: Modular black boxes $\mathbf{B}_a$, $\mathbf{B}_b$ for $a, b \in \mathbb{Z}[x_1, ..., x_n]$, $\hat{f}_{\rho,j-1} \in \mathbb{Z}_p[x_1, ..., x_{j-1}](1 \le \rho \le r)$ s.t. monic(sqf($g_j(x_j = \alpha_j)$)) $= \prod_{\rho=1}^{r} \hat{f}_{\rho,j-1}$, $\alpha \in \mathbb{Z}^{n-1}$, a prime p, degree estimates da_i, db_i, dg_i, and $j \ge 2 \in \mathbb{Z}$.

Output: $\hat{f}_{\rho,j} \in \mathbb{Z}_p[x_1, ..., x_j](1 \le \rho \le r)$ s.t. (i) sqf(g_j) $= \prod_{\rho=1}^{r} \hat{f}_{\rho,j}$, and (ii) monic($\hat{f}_{\rho,j}(x_j = \alpha_j)$) $= \hat{f}_{\rho,j-1} \bmod p\,(1 \le \rho \le r)$ or FAIL.

Let $\hat{f}_{\rho,j-1} = \sum_{i=0}^{df_\rho} \sigma_{\rho,i}(x_2, ..., x_{j-1}) x_1^i$ where $df_\rho = \deg(\hat{f}_{\rho,j-1}, x_1)$ for $1 \le \rho \le r$.
Let $\sigma_{\rho,i} = \sum_{k=1}^{s_{\rho,i}} c_{\rho,ik} M_{\rho,ik}$ where $M_{\rho,ik}$ are monomials in $x_2, ..., x_{j-1}$ and $s_{\rho,i} = \#\sigma_{\rho,i}$.
Pick non-zero $\beta_2, ..., \beta_{j-1} \in \mathbb{Z}_p$ at random.
$\mathcal{S}_{\rho,i} \leftarrow \{m_{\rho,ik} = M_{\rho,ik}(\beta_2, ..., \beta_{j-1})$ for $1 \le k \le s_{\rho,i}\}$ for $1 \le \rho \le r, 0 \le i \le df_\rho$.
if any $|\mathcal{S}_{\rho,i}| \ne s_{\rho,i}$ **return** FAIL.
Let s_j be the maximum of $s_{\rho,i}$.
// Compute s_j images of the factors in $\mathbb{Z}_p[x_1, x_j]$.
$d_1, d_2 \leftarrow dg_j, dg_j$.
Pick $\gamma_0, ..., \gamma_{d_1+d_2}$ unique points at random from $\mathbb{Z}_p$.
for k from 1 to s_j **do**
  Let $Y_k = (x_2 = \beta_2^k, ..., x_{j-1} = \beta_{j-1}^k)$.
  for i from 0 to $d_1 + d_2$ **do**
    Interpolate $A_{ki} = a(x_1, Y_k, \gamma_i, \alpha_{j+1}, ..., \alpha_n) \in \mathbb{Z}_p[x_1]$ via probes to $\mathbf{B}_a$.
    if $\deg(A_{ki}, x_1) \ne da_1$ **return** FAIL.
    Interpolate $B_{ki} = b(x_1, Y_k, \gamma_i, \alpha_{j+1}, ..., \alpha_n) \in \mathbb{Z}_p[x_1]$ via probes to $\mathbf{B}_b$.
    if $\deg(B_{ki}, x_1) \ne db_1$ **return** FAIL.
    $G_{ki} \leftarrow \gcd(A_{ki}, B_{ki}) \in \mathbb{Z}_p[x_1]$.
    if $\deg(G_{ki}, x_1) \ne dg_1$ **return** FAIL.
    $S_{ki} \leftarrow \gcd(G_{ki}, \partial G_{ki}/\partial x_1) \in \mathbb{Z}_p[x_1]$.
    $G_{sf,i} \leftarrow \mathrm{quo}(G_{ki}, S_{ki}) \in \mathbb{Z}_p[x_1]$. // $G_{sf,i}$ = sqf(G_{ki}) mod p, up to a constant in $\mathbb{Z}_p$.
    if $\deg(G_{sf,i}, x_1) \ne \sum_{\rho=1}^{r} df_\rho$ **return** FAIL.
  end
  Interpolate $G_{sf} \in \mathbb{Z}_p[x_1, x_j]$ s.t. $G_{sf}(x_1, \gamma_i) = G_{sf,i}$ for $0 \le i \le d_1 + d_2$.
  $d_{sf} \leftarrow \deg(G_{sf}, x_1)$.
  if $d_2 > 0$ **then**
    $\hat{M} = \prod_{i=0}^{d_1+d_2}(x_j - \gamma_i)$
    Compute the rational function N_i/D_i s.t. $N_i/D_i \equiv \mathrm{coeff}(G_{sf}, x_1, i) \bmod \hat{M}$ for $0 \le i < d_{sf}$.
    $L \leftarrow$LCM$(D_0, ..., D_{d_{sf}-1}) \in \mathbb{Z}_p[x_j]$.
    $G_{sf} \leftarrow L x_1^{d_{sf}} + \sum_{i=0}^{d_{sf}-1} \frac{LN_i}{D_i} x_1^i$.
    $d_1, d_2 \leftarrow \max(\deg(N_0, x_j), ..., \deg(N_{d_{sf}-1}, x_j)), \max(\deg(D_0, x_j), ..., \deg(D_{d_{sf}-1}, x_j))$.
  end
  $G_{sfm} \leftarrow \mathrm{monic}(G_{sf}) \bmod p$. // make G_{sf} monic in x_j.
  $F_{\rho,k} \leftarrow \hat{f}_{\rho,j-1}(x_1, Y_k) \in \mathbb{Z}_p[x_1]$ for $1 \le \rho \le r$.
  if any $\deg(F_{\rho,k}) < df_\rho$ (for $1 \le \rho \le r$) **return** FAIL.
  if $\gcd(F_{\rho,k}, F_{\phi,k}) \ne 1$ for any $1 \le \rho < \phi \le r$ **return** FAIL.
  $\bar{f}_{\rho,k} \leftarrow$ BivariateHenselLift$(G_{sfm}, [F_{1,k}, ..., F_{r,k}], \alpha_j, p)$.
end
if $j = 2$ **return** $[\bar{f}_{1,1}, ..., \bar{f}_{r,1}]$.
Let $\bar{f}_{\rho,k} = \sum_{l=1}^{t_\rho} \alpha_{\rho,kl} \tilde{M}_{\rho,l}(x_1, x_j) \in \mathbb{Z}_p[x_1, x_j]$ for $1 \le k \le s_j$, for $1 \le \rho \le r$ where $t_\rho = \#\bar{f}_{\rho,k}$.
for ρ from *1* to r **do**
  for l from *1* to t_ρ **do**
    $i \leftarrow \deg(\tilde{M}_{\rho,l}, x_1)$.
    Solve the linear system $\{ \sum_{k=1}^{s_{\rho,i}} m_{\rho,ik}^t c_{\rho,lk} = \alpha_{\rho,tl}$ for $1 \le t \le s_{\rho,i}$ $\}$ for $c_{\rho,lk} \in \mathbb{Z}_p$.
    $T_l \leftarrow \sum_{k=1}^{s_{\rho,i}} c_{\rho,lk} M_{\rho,ik}(x_2, ..., x_{j-1})$.
  end
  $\hat{f}_{\rho,j} \leftarrow \sum_{l=1}^{t_\rho} T_l \tilde{M}_{\rho,l}(x_1, x_j)$.
end
return $\hat{f}_{\rho,j}(1 \le \rho \le r)$

4 Complexity Analysis

Let $g = \gcd(a, b)$. In this section, we determine the complexity of Algorithm 2: MHL_BB_PGCD to compute the square-free factors $f_1, f_2, ..., f_r$ of $\text{pp}(g, x_1)$. We do not include the cost of computing the square-free factors of $\text{cont}(g, x_1)$. The complexity is given in terms of the size of the inputs a and b and the size of the outputs $f_1, f_2, ..., f_r$. Throughout the analysis $d_i = \max(\deg(a, x_i), \deg(b, x_i))$ for $1 \le i \le n$, $D = \max(d_1, ..., d_n)$, C_a and C_b are the number of arithmetic operations in $\mathbb{Z}_p$ to evaluate $\mathbf{B}_a$ and $\mathbf{B}_b$ respectively and $\#F = \sum_{\rho=1}^{r} \#f_\rho$ is the number of terms in the square-free factors. In rare cases, the number of terms of the output can be larger than the number of terms in the input.

Algorithm CM_BB_SHL_GCD calls algorithm CM_BB_SHL_GCD_stepj $n-1$ times to recover $x_2, x_3, ..., x_n$ one at a time in the square-free factors of $\text{pp}(g, x_1)$. This means we lose a factor of $n-1$ in efficiency when compared with algorithms like Kaltofen-Diaz which can interpolate all variables in g simultaneously. The core of Algorithm CM_BB_SHL_GCD_stepj is steps 9 to 36 whose cost is multiplied by s_j. Let $df_\rho = \deg(f_\rho, x_1)$ and let $f_\rho = \sum_{i=0}^{df_\rho} \tau_{\rho,i}(x_2, ..., x_n) x_1^i$. Step 2 defines

$$s_{\rho,i} = \#\sigma_{\rho,i}(x_2, ..., x_{j-1}) = \#\tau_{\rho,i}(x_2, ..., x_{j-1}, \alpha_j, ..., \alpha_n).$$

Let s_j be the the maximum number of terms in any of the coefficients $\sigma_{\rho,i}$ at step j, that is, let $s_j = \max_{\rho,i} s_{\rho,i}$ and let $s_T = \sum_{j=2}^{n} s_j$. The number of terms in $\sigma_{\rho,i}$ will only increase with subsequent calls to Algorithm 4, thus $s_2 \le s_3 \le \cdots \le s_n$. It follows that $s_T \le (n-1)s_n$ and $s_n \le \max_{\rho,i} \#\tau_{\rho,i}$. The ratio $\#g/s_n$ represents a speedup of our algorithm over an algorithm that interpolates g.

In Example 1, we have $\text{pp}(g, x_1) = f_1 f_2^2$ where $f_1 = x_1 + x_3$ and $f_2 = x_1 - x_2$. In this example, we have $s_2 = s_3 = 1$ and $\#g = 10$. In Example 2, we have $\text{pp}(g, x_1) = f_1 f_3^3$ where $f_1 = x_1 + 2x_2 + \frac{1}{2}$ and $f_3 = x_1 - x_3$, so $s_2 = 1, s_3 = 2$ and $\#g = 21$.

4.1 CM_BB_SHL_GCD Complexity

We give the following theorem for the complexity of Algorithm 3: CM_BB_SHL_GCD.

Theorem 1. *Let p be a large prime and, $a, b \in \mathbb{Z}[x_1, ..., x_n]$. If Algorithm CM_BB_SHL_GCD does not return FAIL, then the total number of arithmetic operations in $\mathbb{Z}_p$ for lifting $\hat{f}_{\rho,1}$ to $\hat{f}_{\rho,n}$ using Algorithm CM_BB_SHL_GCD_stepj is at most*

$$O\left(s_T\left(d_1 D(d_1 + D + C_a + C_b) + (n + D)\#F + nD\right)\right). \tag{2}$$

From (2), one sees that the total number of probes to the black boxes is $O(s_T d_1 D)$.

Proof. Let $da_j = \deg(a, x_j)$, $db_j = \deg(b, x_j)$, $dg_j = \deg(g, x_j)$, and $\tilde{d}_j = \deg(\text{sqf}(\gcd(a, b)), x_j)$ for $1 \le j \le n$. In step 12 of Algorithm 4, we use dense interpolation to get the univariate image $a(x_1, Y_k, \gamma_i, \alpha_{j+1}, ..., \alpha_n)$. This operation does $O(da_1)$ probes to $\mathbf{B}_a$ and $O(da_1^2)$ arithmetic operations in $\mathbb{Z}_p$. The total cost of step 12 for the s_j interpolations is $O(s_j da_1 dg_j C_a) + O(s_j da_1^2 dg_j) \subseteq O(s_j d_1 d_j C_a) + O(s_j d_1^2 d_j)$ arithmetic operations in $\mathbb{Z}_p$ since $d_j \ge da_j \ge dg_j$ for $1 \le j \le n$. Similarly, the total cost of step 14 is $O(s_j db_1 dg_j C_b) + O(s_j db_1^2 dg_j) \subseteq O(s_j d_1 d_j C_b) + O(s_j d_1^2 d_j)$.

For step 16, we use the Euclidean algorithm to compute the GCD of A_{ki} and B_{ki} in $\mathbb{Z}_p[x_1]$. The Euclidean algorithm does $O(d_1^2)$ arithmetic operations in $\mathbb{Z}_p$. The total cost of step 16 is $O(s_j d_1^2 d_j)$ arithmetic operations in $\mathbb{Z}_p$.

For step 18, we again use the Euclidean algorithm which does $O(dg_1^2)$ arithmetic operations in $\mathbb{Z}_p$. The division in $\mathbb{Z}_p[x_1]$ in step 19 can also be done with $O(dg_1^2)$ arithmetic operations. Since $dg_1 \leq d_1$ and $dg_j \leq d_j$ the total cost of steps 18 and 19 is $O(s_j d_1^2 d_j)$ arithmetic operations in $\mathbb{Z}_p$.

For step 25, the polynomial $\hat{M} = \prod_{i=0}^{d_1+d_2}(x_j - \gamma_i)$ needs to be expanded. Expanding $\hat{M}$ one factor at a time uses $O(dg_j^2)$ arithmetic operations in $\mathbb{Z}_p$ as $d_1 + d_2 \leq 2dg_j$. The total cost of step 25 is $O(s_j d_j^2)$.

Step 26 performs rational function interpolation on each of the polynomial coefficients of G_{sf}. Rational function interpolation uses the extended Euclidean algorithm (see Section 5.7 in [8]) to compute the rational functions N_i/D_i for $0 \leq i < d_{sf}$ where each $N_i, D_i \in \mathbb{Z}_p[x_j]$. One application of rational function interpolation does $O(dg_j^2)$ arithmetic operations in $\mathbb{Z}_p$. Since $dg_j \leq d_j$ and $d_{sf} \leq dg_1 \leq d_1$, the total cost of step 26 is $O(s_j d_1 d_j^2)$.

Step 27 computes the least common multiple (LCM) of the polynomials D_i for $0 \leq i < d_{sf}$ where $\deg(D_i, x_j) \leq dg_j$. We compute $L = \text{LCM}(D_0, D_1, ..., D_{d_{sf}-1}) \in \mathbb{Z}_p[x_j]$ by first computing $\text{LCM}(D_0, D_1)$, then $\text{LCM}(\text{LCM}(D_0, D_1), D_2)$ and so on until we compute the least common multiple of all $d_{sf} - 1$ polynomials. As L is the leading coefficient of G_{sf}, its degree, as well as the degree of any of the intermediate polynomials produced, is at most dg_j. We compute the LCM of two polynomials using the formula $\text{LCM}(a, b) = ab/\gcd(a, b)$ for polynomials $a, b \in \mathbb{Z}_p[x]$. Clearly, any one LCM computation does $O(dg_j^2)$ arithmetic operations in $\mathbb{Z}_p$ as $\deg(L, x_j) \leq dg_j$. Since $dg_j \leq d_j$ and $d_{sf} \leq dg_1 \leq d_1$, the total cost of step 27 is $O(s_j d_1 d_j^2)$.

For step 28, we perform the division L/D_i for $0 \leq i < d_{sf}$. Each division does $O(dg_j^2)$ arithmetic operations in $\mathbb{Z}_p$. So, the total cost of step 28 is $O(s_j d_1 d_j^2)$.

Step 32 evaluates $\hat{f}_{\rho,j-1}(x_1, \beta_2^k, ..., \beta_{j-1}^k)$ for $1 \leq \rho \leq r$. If we first compute the powers of β_i^k using $\sum_{i=2}^{j-1} dg_i \leq (j-2)D$ multiplications, we can evaluate the terms in the factors $\hat{f}_{\rho,j-1}$ using $(j-2)\sum_{\rho=1}^{r} \#\hat{f}_{\rho,j-1}$ multiplications. Since $j - 2 < n$ and $\#\hat{f}_{\rho,j-1} \leq \#f_\rho$, the total cost of step 32 is $O(s_j nD + s_j n \sum_{\rho=1}^{r} \# f_\rho)$ arithmetic operations in $\mathbb{Z}_p$.

For step 35, we use Monagan and Paluck's bivariate Hensel lifting algorithm from [16,17] which does $O(\tilde{d_1}^2 \tilde{d_j} + \tilde{d_1}\tilde{d_j}^2)$ arithmetic operations in $\mathbb{Z}_p$. The total cost of step 35 is $O(s_j(\tilde{d_1}^2 \tilde{d_j} + \tilde{d_1}\tilde{d_j}^2)) \subseteq O(s_j(d_1^2 d_j + d_1 d_j^2))$.

Using Zippel's algorithm [24], the cost of solving the Vandermonde system in step 42 for the coefficients of any given factor $\hat{f}_{\rho,j-1}$ is $\tilde{d}_j \sum_{i=0}^{df_\rho - 1} O(s_{\rho,i}^2) \subseteq O(\tilde{d}_j s_j \#\hat{f}_{\rho,j-1})$ since $\sum_{i=0}^{df_\rho-1} s_{\rho,i} < \#\hat{f}_{\rho,j-1}$. The total cost to solve for the coefficients of all factors $\hat{f}_{\rho,j}$ is $O(s_j \tilde{d}_j \sum_{\rho=1}^{r} \#\hat{f}_{\rho,j-1}) \subseteq O(s_j d_j \sum_{\rho=1}^{r} \#f_\rho)$.

Summing the costs, the total number of arithmetic operations in $\mathbb{Z}_p$ for Algorithm CM_BB_SHL_GCD_stepj to recover x_j is

$$O(s_j(d_1^2 d_j + d_1 d_j^2 + d_1 d_j(C_a + C_b) + (n + d_j)\sum_{\rho=1}^{r} \#f_\rho + nD)). \tag{3}$$

Since $d_j \leq D$, $s_T = \sum_{j=2}^{n} s_j$, and $\sum_{\rho=1}^{r} \#f_\rho = \#F$, summing (3) for $j = 2, 3, ..., n$ gives (2).

4.2 MHL_BB_PGCD Complexity

The following theorem gives the complexity of Algorithm 2: MHL_BB_PGCD for computing $\text{monic}(\text{pp}(\gcd(a, b), x_1))$.

Theorem 2. *Let p be a large prime and let $a, b \in \mathbb{Z}[x_1, ..., x_n]$ and let $g = \text{monic}(\text{pp}(\gcd(a, b), x_1))$ mod p. If algorithm MHL_BB_PGCD does not return FAIL, the total number of arithmetic operations in $\mathbb{Z}_p$ in the worst case for computing g using Algorithm MHL_BB_PGCD is*

$$O\left(s_T\left(D^2(D + C_a + C_b) + (n + D)\#F + nD\right)\right). \tag{4}$$

From (4) the number of probes to $\boldsymbol{B}_a$ and $\boldsymbol{B}_b$ is $O(s_T D^2)$.

Proof. Step 2 of Algorithm MHLBBGCD makes $O(da_1)$ probes to $\mathbf{B}_a$ and does $O(da_1^2)$ arithmetic operations in $\mathbb{Z}_p$ to interpolate a_1 in $\mathbb{Z}_p[x_1]$. Similarly, step 4 makes $O(db_1)$ probes to $\mathbf{B}_b$ and does $O(db_1^2)$ arithmetic operations in $\mathbb{Z}_p$ to interpolate b_1 in $\mathbb{Z}_p[x_1]$. In step 6, the Euclidean algorithm does $O(da_1db_1)$ arithmetic operations in $\mathbb{Z}_p$ to compute g_1 and step 8 needs $O(da_1db_1)$ arithmetic operations in $\mathbb{Z}_p$ to compute the square-free factorization of g_1 (see [8]). These costs are dominated by the cost of Algorithm CM_BB_SHL_GCD in step 10 which does $O(s_T(d_1D(d_1+D+C_a+C_b)+(n+D)\#F + nD))$ arithmetic operations in $\mathbb{Z}_p$ by Theorem 1. The theorem follows since $d_1 \le D$.

5 Benchmarks

We present three timing benchmarks with each benchmark executed in Maple 2024. All timings were obtained using one core on a server with 128 gigabytes of RAM and two Intel Xeon E5-2680 processors running at 2.80GHz base and 3.60GHz turbo.

Let $a, b \in \mathbb{Z}[x_1, ..., x_n]$, $g = \gcd(a, b)$, $c = a/g$ and $d = b/g$. For comparison, we've created two additional algorithms which compute monic(g). For the first algorithm, we create a new modular black box for computing $f = a/b$ and use the sparse rational function interpolation algorithm proposed by Kaltofen and Yang [14], which outputs black boxes, for computing $c(\sigma)$ and $d(\sigma)$ for a given point $\sigma \in \mathbb{Z}_p^n$. From this we can compute $g(\sigma) = a(\sigma)/c(\sigma)$. We combined this method with Ben-Or/Tiwari sparse interpolation [1] to interpolate g mod p. We then used our BB_MGCD algorithm to find monic(g) using Chinese remaindering and rational number reconstruction.

For the second algorithm, we modify the black box GCD algorithm proposed by Kaltofen and Diaz in [7] to make it into a modular algorithm. Kaltofen and Diaz's algorithm creates a black box $\mathbf{B}_g$ such that $\mathbf{B}_g(\sigma)$ computes $g(\sigma)$. Instead, we create a modular black box for computing $g(\sigma)$ mod p and combine the modular black box with Ben-Or/Tiwari sparse interpolation and our BB_MGCD algorithm to compute monic(g), again using Chinese remaindering and rational number reconstruction. We shall refer to these new algorithms as the "Kaltofen–Yang" algorithm and "Kaltofen–Diaz" algorithm respectively. For both the Kaltofen–Yang and Kaltofen–Diaz algorithms, we randomize the input as follows. We pick $\beta \in \mathbb{Z}_p^n$ and construct a new black box $\mathbf{B}_c : (\mathbb{Z}^n, p) \to \mathbb{Z}_p$ where $\mathbf{B}_c(\alpha, p) = \mathbf{B}_a([\beta_1\alpha_1, \beta_2\alpha_2, ..., \beta_n\alpha_n], p)$ to avoid evaluating a at an unlucky point w.h.p.

5.1 Benchmark 1

Our first benchmark is taken from Kaltofen-Diaz [7]. Let

$$V_1 = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix}.$$

V_1 is an $n \times n$ Vandermonde matrix in the variables $[x_1, x_2, ..., x_n]$. Let V_2 be an $n \times n$ Vandermonde matrix in $[x_1, ..., x_{n/2}, y_{n/2+1}, ..., y_n]$. We create two modular black boxes $\mathbf{B}_a$, $\mathbf{B}_b : (\mathbb{Z}^n, p) \to \mathbb{Z}_p$ such that $\mathbf{B}_a$ and $\mathbf{B}_b$ return the determinants of V_1 and V_2 respectively evaluated at a point modulo a prime p. We compute the GCD of $\mathbf{B}_a$ and $\mathbf{B}_b$, namely

$$\gcd(\det(V_1), \det(V_2)) = \prod_{i=1}^{n/2-1} \prod_{j=i+1}^{n/2} (x_i - x_j). \tag{5}$$

Table 1 shows the CPU timings (in seconds) for our new algorithm compared against the Kaltofen-Yang (KY) and Kaltofen-Diaz (KD) algorithms to compute (5). All algorithms use the two primes $p = 2^{61} + 15$ and $p = 2^{61} + 21$ to compute the GCD. Column $\#g$ is the number of terms in g, column $\#\text{pp}(g, x_1)$ is the number of terms in $\text{pp}(g, x_1)$, column s_n is the largest s_j used by Algorithm 4, $s_T = \sum_{j=2}^{n} s_j$ used in Algorithm 4, and column #probes is the number of times each algorithm had to probe a black box. Column "eval BB" is the time (in seconds) used by Algorithm 4 to probe the black boxes prior to univariate interpolation (steps 12 and 14). For our BB_MGCD algorithm, the timings given in brackets are the time needed to compute $\text{pp}(\gcd(\det(V_1), \det(V_2)), x_1)$ only. This illustrates the advantage of not computing $\text{cont}(g, x_1)$.

					BB_MGCD			KY		KD	
n	$\#g$	$\#\text{pp}(g, x_1)$	s_n	s_T	time (pp only)	eval BB	#probes	time	#probes	time	#probes
4	2	2	1	6 (6)	0.061 (0.061)	0.00 (0.00)	766 (766)	0.039	408	0.042	264
6	6	4	2	20 (29)	0.205 (0.126)	0.04 (0.01)	3646 (2446)	0.076	1750	0.100	1090
8	24	8	3	41 (79)	0.600 (0.251)	0.18 (0.04)	10742 (5594)	0.289	10860	0.592	6828
10	120	16	6	102 (183)	1.522 (0.530)	0.60 (0.10)	27842 (13702)	2.122	72590	4.503	46142
12	720	32	10	203 (438)	3.506 (0.952)	1.67 (0.17)	63918 (28882)	23.438	729936	56.967	468120
14	5040	64	20	468 (969)	7.815 (2.024)	4.18 (0.50)	146654 (68726)	FAIL	-	FAIL	-
16	40320	128	35	929 (2027)	18.171 (4.244)	10.61 (1.02)	320790 (146026)	FAIL	-	FAIL	-
18	362880	256	70	2070 (4329)	45.889 (9.981)	29.15 (2.64)	722354 (346822)	FAIL	-	FAIL	-
20	3628800	512	126	4114 (8883)	118.941 (23.392)	82.14 (5.92)	1577282 (743606)	FAIL	-	FAIL	-

Table 1. Benchmark 1: Timings in CPU seconds

Our algorithm is faster than both the Kaltofen-Yang and Kaltofen-Diaz algorithms when $n \geq 10$. Ben-Or/Tiwari sparse interpolation is relatively fast when interpolating polynomials with a small number of terms, but slows down when interpolating polynomials with many terms. This partially explains why both the Kaltofen-Yang and Kaltofen-Diaz algorithm outperform our BB_MGCD algorithm when n is small. In Table 1, FAIL means the Ben-Or/Tiwari sparse interpolation needed a prime greater than 2^{63} which we have not implemented and it would be slow. When computing $\text{cont}(g, x_1)$, our algorithm spends the largest amount of time on probing the black boxes for interpolations for $n \geq 12$.

5.2 Benchmark 2

Consider the polynomials $a = f_1 f_2^2 \bar{a}$ and $b = f_1 f_2^2 \bar{b}$ in $\mathbb{Z}[x_1, ..., x_8]$ with $g = \gcd(a, b) = f_1 f_2^2$. We generate the polynomial f_1 with t_1 terms where each monomial is chosen randomly from the set of monomials with the indeterminates $x_1, ..., x_8$ that have a total degree of at most 5 and each

integer coefficient is chosen randomly from [-99,99]. We define $f_2, \bar{a}$, and $\bar{b}$ similarly to f_1 except that they have t_2, t_3, and t_4 terms respectively. For all benchmarks, we set $t_3 = t_4 = 10$. We create the modular black boxes $\mathbf{B}_a$ and $\mathbf{B}_b$ which evaluate the polynomials a and b at a point modulo a prime p. We note that g is not square-free which is a best case for our algorithm.

Table 2 uses the same algorithms and terms as Table 1.

					BB_MGCD			KY		KD	
$\#g$	t_1	t_2	s_n	s_T	time	eval BB	#probes	time	#probes	time	#probes
495	10	10	5	44	1.510	0.42	15825	2.110	42194	6.001	76913
4071	20	20	12	59	2.209	0.85	26041	17.374	279537	57.858	597986
27409	40	40	26	97	4.739	1.58	53277	75.142	675546	1008.83	4564903
118031	80	80	49	156	6.672	2.35	72623	133.199	719958	13014.39	19056681

Table 2. Benchmark 2: Timings in CPU seconds

Our BB_MGCD algorithm outperforms the Kaltofen–Yang and Kaltofen–Diaz algorithms in all cases. As our algorithm only has to lift to sqf(g) when calling the CM_BB_SHL_GCD algorithm, it does significantly fewer probes to $\mathbf{B}_a$ and $\mathbf{B}_b$ than the Kaltofen-Yang and Kaltofen-Diaz algorithms. As stated before, Benchmark 2 is a best case for our algorithm.

5.3 Benchmark 3

Our third benchmark is taken from Monagan and Huang [12]. We want to calculate the GCD of two polynomials with $n = 8$ variables. We create the polynomial g with s terms and polynomials c and d with t terms where each monomial is chosen randomly from the set of monomials with a total degree of at most 12 and each integer coefficient is chosen randomly from $[-99, 99]$. We create the modular black boxes $\mathbf{B}_a$ and $\mathbf{B}_b$ which evaluate the polynomials $a = gc$ and $b = gd$ at a point modulo a prime p. Since g, c, d are created randomly, we have $g = \gcd(a, b)$ where g is square-free and has no polynomial content which is a worst case for our algorithm.

Table 3 uses the same algorithms and terms as Table 1.

				BB_MGCD		KY		KD	
s	t	s_n	s_T	time	#probes	time	#probes	time	#probes
10	10000	5	28	11.388	20445	1.905	3337	1.911	3169
100	5000	42	187	39.132	131231	7.359	24757	7.679	23400
1000	2500	407	1303	270.577	1193566	46.589	215176	50.064	203032
2500	1000	943	2589	626.824	2474626	152.015	699817	166.754	660239
5000	100	1925	4398	1756.228	4222639	428.820	1398957	454.156	1319793
10000	10	3619	7320	5422.783	5606030	1589.731	2797277	1635.116	2638956

Table 3. Benchmark 3: Timings in CPU seconds

We see a different result from this benchmark. The Kaltofen–Yang and Kaltofen–Diaz algorithms both outperform our BB_MGCD algorithm in all cases. Part of the reason for this is that

the Kaltofen–Yang and Kaltofen–Diaz algorithms use Ben-Or/Tiwari sparse interpolation which recovers all variables in g simultaneously. However, notice that our algorithm is gaining as $s = \#g$ increases.

In Theorem 1, the number of probes is multiplied by a factor of $s_T \leq (n-1)s_n$. The factor of $n-1$ is because our algorithm recovers the variables $x_2, \ldots, x_n$ in g one at a time using Hensel lifting. In comparison, the other algorithms use BenOr/Tiwari interpolation, which recovers all variables in g simultaneously. But s_n can be significantly smaller than $\#g$ which represents a gain of a factor of $\#g/s_n$ for our algorithm.

If $g = \sum_{i=0}^{d_1} a_i(x_2, \ldots, x_n)x_1^i$, for this benchmark, when we recover x_n using Hensel lifting, $s_n = \max_{i=0}^{d_1} \#a_i(x_2, \ldots, x_{n-1}, \alpha_n) \leq \max_{i=0}^{d_1} \#a_i$, that is, s_n is at most the number of terms of the largest coefficient of g in x_1. The terms in g are unlikely to be equally distributed among the a_i coefficients. The largest value for s_n (the worst case for our algorithm) occurs when $g = x_1^{d_1} + a_0(x_2, \ldots, x_n)$, that is, only one term involves x_1. The lowest value for s_n (the best case for our algorithm) occurs when the terms of g are distributed equally among the coefficients $a_0, a_1, \ldots, a_{d_1}$. Thus $\#g/(d_1+1) \leq s_n < \#g$ hence $1 < \#g/s_n \leq d_1 + 1$.

In benchmark 3, we choose the terms of g at random from those of degree at most d. Since a polynomial f in n variables with $\deg(f) = d$ has at most $\binom{n+d}{d}$ terms, the expected value

$$E\left[\frac{\#g}{s_n}\right] = \frac{\binom{n+d}{d}}{\binom{n-1+d}{d}} = \frac{(n+d)}{n}.$$

For benchmark 3 where $n = 8$ and $d = 12$, $E[\#g/s_n] = 2.5$ which is low. It's a lot less than the best case $d + 1 = 13$.

6 Implementation Notes

We have implemented our new black box GCD algorithm in Maple with some subroutines coded in C. Our code is available for download at http://www.cecm.sfu.ca/~mmonagan/code/BBMGCD/

For Algorithm 4 CM_BB_SHL_GCD_stepj we have implemented the univariate interpolations in steps 12 and 14 in C. In Algorithm 4 we use Maple's GCD algorithm to compute $\gcd(a_1, b_1)$ in $\mathbb{Z}_p[x_1]$ which is coded in C. The bivariate Hensel lift in step 35 of Algorithm 4 is coded in C. We use our algorithm from [17]. For step 42 in Algorithm 4, we solve a Vandermonde system of dimension $s_{\rho,i}$. We coded Zippel's algorithm [24] in C. It does $O(s_{\rho,i}^2)$ arithmetic operations in $\mathbb{Z}_p$. We note that the main for loop in line 9 of Algorithm 4 CM_BB_SHL_GCD_stepj can be parallelized.

We have performed several optimizations to our black box implementations in Benchmark 1 in order to speed up the black box probes. Our black boxes compute $\det(V_1)$ and $\det(V_2)$ evaluated at a point $\alpha \in \mathbb{Z}_p^{2n}$ using the formula $\det(V_1) = \prod_{i=1}^{n-1} \prod_{j=i+1}^{n} (x_i - x_j)$ and similarly for $\det(V_2)$. For Benchmark 2, we do not expand the polynomials $A = CG$ and $B = DG$ in our black boxes. Instead, we evaluate C, D and G independently, then return the products.

Two places where we need to evaluate polynomials in $\mathbb{Z}_p[x_1, ..., x_n]$ are the partial factors in step 32 of Algorithm 4 and the polynomial $\hat{f}$ in steps 16 and 17 of Algorithm 2. We perform these multivariate polynomial evaluations modulo a 62 bit prime p using a C program for efficiency. Since Maple has two representations for polynomials in $\mathbb{Z}[x_1, ..., x_n]$, namely, the old SUM-OF-PROD representation and the new POLY representation (see [18]), we must handle both representations. Even with these evaluations coded in C, often more than 50% of the time is spent in these evaluations on our benchmarks. Table 4 gives a timing breakdown for the main steps of Algorithm 4 for

benchmark 1 for $n = 20$. It shows that $82.14/118.94 = 69\%$ of the total time was spent in black box probes.

Operation	time(s)
Compute monomial evaluations (step 4)	1.01
Probe $\mathbf{B}_a$ and $\mathbf{B}_b$ for interpolation (steps 12,14)	82.14
Perform univariate interpolations (steps 12,14)	14.39
Compute univariate GCD (step 16)	3.46
Compute 2nd univariate GCD (step 18)	2.60
Rational Function Reconstruction (step 26)	0.40
Evaluate $\hat{f}_{\rho,j-1}(x_1, Y_k)$ (step 32)	1.69
Perform BivariateHenselLift (step 35)	0.43
Solve Vandermonde systems (step 42)	6.42
Other operations	6.40
Total	118.94

Table 4. Algorithm 3 breakdown for Benchmark 1 for $n = 20$

7 Conclusion

In this paper, we have contributed a new algorithm for computing the multivariate GCD of sparse polynomials represented by black boxes. Our algorithm constructs a factorization of the GCD g from a sequence of bivariate images of g. We also constructed a black box GCD algorithm from Kaltofen-Yang's sparse rational function interpolation algorithm. Our algorithm was much faster than the Kaltofen-Yang and Kaltofen-Diaz black box GCD algorithms on two of the three benchmarks but slower on the other benchmark. We gave a complexity analysis for our new algorithm but have yet to complete a failure probability analysis.

We designed our algorithm to interpolate the square-free factors of $g = \gcd(a, b)$ which gives it an advantage when the square-free factors are smaller than g. We could design it to instead recover the irreducible factors of g over $\mathbb{Z}$ by lifting a factorization of $g(x_1, \alpha)$ over $\mathbb{Z}$. This would be faster for benchmark 1 where the factors all have 2 terms. We chose not to do this because of the additional cost of a factorization in $\mathbb{Z}[x]$ and because it makes the algorithm more complicated. Computing a square-free factorization is easy and does not increase the cost.

Acknowledgments. This work was supported by the National Science and Research Council of Canada (NSERC) and Maplesoft.

Disclosure of Interests. Michael Monagan has received a research grant from Maplesoft.

References

1. Ben-Or, M., and Tiwari, P. A Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation. In *Proceedings of STOC '88* (1988), ACM, pp. 301–309.
2. Brown, W., and Traub, J. On Euclid's Algorithm and the Theory of Subresultants. *J. ACM* ***18*** (1971), 505–514.

3. Brown, W. S. On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors. *J. ACM* ***18***, 4 (1971), 478–504.
4. Chen, T. *Sparse Hensel Lifting Algorithms for Multivariate Polynomial Factorization.* PhD thesis, Simon Fraser University, 2024.
5. Chen, T., and Monagan, M. A New Black Box Factorization Algorithm - the Non-Monic Case. In *Proceedings of ISSAC '23* (2023), ACM, p. 173–181.
6. Cuyt, A., and Lee, W.-s. Sparse interpolation of multivariate rational functions. *Theoretical Computer Science* ***412***, 16 (2011), 1445–1456.
7. Díaz, A., and Kaltofen, E. On Computing Greatest Common Divisors with Polynomials given by Black Boxes for Their Evaluations. In *Proceedings of ISSAC '95* (1995), ACM, p. 232–239.
8. Gathen, J. v. z., and Jürgen, G. *Modern Computer Algebra*, 3 ed. Cambridge University Press, 2013.
9. Geddes, K. O., Czapor, S. R., and Labahn, G. *Algorithms for Computer Algebra.* Kluwer Academic, 1992.
10. Giorgi, P., Grenet, B., Perret du Cray, A., and Roche, D. Sparse polynomial interpolation and division in soft-linear time. In *Proceedings of ISSAC '22* (2022), ACM, pp. 459–468.
11. Hu, J., and Monagan, M. A fast parallel sparse polynomial gcd algorithm. In *Proceedings of ISSAC '16* (2016), ACM, pp. 271–278.
12. Huang, Q.-L., and Monagan, M. A New Sparse Polynomial GCD Algorithm by Separating Terms. In *Proceedings of ISSAC '24* (2024), ACM, pp. 134–142.
13. Kaltofen, E., and Trager, B. M. Computing with Polynomials Given by Black Boxes for Their Evaluations: Greatest Common Divisors, Factorization, Separation of Numerators and Denominators. *J. Symb. Comput.* ***9***, 3 (1990), 301–320.
14. Kaltofen, E., and Yang, Z. On Exact and Approximate Interpolation of Sparse Rational Functions. In *Proceedings of ISSAC '07* (2007), ACM, p. 203–210.
15. Monagan, M. Maximal Quotient Rational Reconstruction: An Almost Optimal Algorithm for Rational Reconstruction. In *Proceedings of ISSAC '04* (2004), ACM, p. 243–249.
16. Monagan, M. Linear Hensel Lifting for $\mathbb{Z}_p[x, y]$ and $\mathbb{Z}[x]$ with Cubic Cost. In *Proceedings of ISSAC '19* (2019), ACM, pp. 299–306.
17. Monagan, M., and Paluck, G. Linear Hensel Lifting for $\mathbb{Z}_p[x, y]$ for n Factors with Cubic Cost. In *Proceedings of ISSAC '22* (2022), ACM, p. 159–166.
18. Monagan, M., and Pearce, R. The design of Maple's sum-of-products and POLY data structures for representing mathematical objects. *Commun. Comput. Algebra* ***48***, 3/4 (2014), 160–186.
19. van der Hoeven, J., and Lecerf, G. On sparse interpolation of rational functions and gcds. *Commun. Comput. Algebra* ***55***, 1 (2021), 1–12.
20. van der Hoeven, J., and Lecerf, G. Fast interpolation of multivariate polynomials with sparse exponents. *J. Complex.* ***87***, C (2025).
21. Wang, P. S. The EEZ-GCD Algorithm. *SIGSAM Bull.* ***14***, 2 (1980), 50–60.
22. Wang, P. S., Guy, M. J. T., and Davenport, J. H. P-adic Reconstruction of Rational Numbers. *SIGSAM Bull.* ***16***, 2 (1982), 2–3.
23. Zippel, R. Probabilistic Algorithms for Sparse Polynomials. In *Proceedings of ISSAC '79* (1979), Springer-Verlag, p. 216–226.
24. Zippel, R. Interpolating Polynomials from their Values. *J. Symb. Comput.* ***9***, 3 (1990), 375–403.