Speeding up polynomial GCD, a crucial operation in Maple

Michael Monagan, Simon Fraser University
Email: mmonagan@sfu.ca

Abstract

Given two multivariate polynomials A and B with integer coefficients we present a new GCD algorithm which computes $G = \gcd(A, B)$. Our algorithm is based on the Hu/Monagan GCD algorithm. If $A = G\bar{A}$ and $B = G\bar{B}$ we have modified the Hu/Monagan so that it can interpolate the smaller of G and $\bar{A}$.

We have implemented the new GCD algorithm in Maple with several subroutines coded in C for efficiency. Maple currently uses Zippel's sparse modular GCD algorithm. We present timing results comparing Maple's implementation of Zippel's algorithm with our new algorithm. The new algorithm is much faster on our benchmarks.

1 Introduction

Let $A, B \in \mathbb{Z}[x_0, x_1, \ldots, x_n]$. That is, A and B are polynomials in $n+1$ variables with integer coefficients. In this work we present a Maple implementation of a new algorithm that computes G the greatest common divisor (GCD) of A and B, that is, $G = \gcd(A, B)$. Let $\bar{A} = A/G$ and $\bar{B} = B/G$. We refer to the polynomials $\bar{A}$ and $\bar{B}$ as the cofactors of A and B respectively. The Maple command

```
> simplify(A/B);
```

computes $G = \gcd(A, B)$ then outputs the simplified fraction

$$\frac{A/G}{B/G} = \frac{\bar{A}}{\bar{B}}.$$

How important is the GCD operation for a computer algebra system like Maple? We argue that the speed of the algorithm used to compute G is critical to the overall speed of Maple. If Maple is computing with fractions of polynomials Maple will do many polynomial additions, multiplications, divisions and GCDs. Because GCD is by far the most expensive of these four operations, it will be the bottleneck of the computation in general. What makes GCD computation difficult is that the Euclidean algorithm blows up when it is applied to multivariate polynomials; it is exponential in the number of variables.

In practice, because polynomials in many variables are usually sparse, we want our GCD algorithm to be efficient for sparse polynomials.

Definition 1 (Sparse Polynomial)**.** Let f be a polynomial in n variables with total degree $d = \deg(f)$. Let $\#f$ denote the number of non-zero terms of f. The maximum number of terms in f is $D = \binom{n+d}{d}$. We say f is *sparse* if $\#f \ll D$.

This standard definition for sparse polynomials is imprecise. We prefer the definition f is sparse if $\#f \leq \sqrt{D}$.

Example 1. The polynomial $f = x_1^4x_2 + x_2^4x_3 + x_3^4x_4 + x_4^4x_5 + x_5^4x_1 + 1$ is sparse. Here $\#f = 6$, $d = 5$, $n = 5$, $\binom{n+d}{d} = \binom{10}{5} = 252$ and $\sqrt{252} = 15.9$

The GCD problem for $A, B \in \mathbb{Z}[x_0, x_1, \ldots, x_n]$ has a long history. The first breakthrough was Brown's modular GCD algorithm from [3]. It solves the problem of the blowup in the Euclidean algorithm but it is a dense algorithm. In [19] Moses and Yun developed a multivariate GCD algorithm based on multivariate Hensel lifting (see Ch. 6 in [5]). They called their algorithm the EZ-GCD algorithm. In [23] Wang improved the EZ-GCD algorithm for sparse polynomials. Wang dubbed his algorithm the EEZ-GCD algorithm. Wang's algorithm was implemented in Maple by Keith Geddes in the 1980s and used as Maple's principle GCD algorithm for polynomials with 3 or more variables until 2005.

Since 2005 Maple has been using Zippel's sparse modular GCD algorithm from [24]. The Maple implementation of Zippel's algorithm is described in [9]. Zippel's algorithm was one of the first probabilistic algorithms. Other computer algebra systems that use Zippel's algorithm include Fermat, Magma, and Mathematica. The algorithm presented in this work is based on Hu and Monagan's GCD algorithm in [7, 8].

Brown's GCD algorithm, Zippel's GCD algorithm, and the Hu/Monagan GCD algorithm all work by interpolating G from univariate images. They all compute $\gcd(A, B)$ modulo a sequence of primes $p_1, p_2, ...$ then apply the Chinese remainder theorem to reconstruct the integer coefficients in G. The algorithms differ in how they compute $G = \gcd(A, B) \bmod p$ for a prime p. They use different interpolation algorithms to interpolate G[1] from univariate images $g_j = G(x_0, \beta_j)$ for $\beta_j \in \mathbb{Z}_p^n$.

Let $d_i = \deg(G, x_i)$, $D = d_1 + d_2 + \cdots + d_n$, and $P = (d_1+1)(d_2+1)\ldots(d_n+1)$. Brown's algorithm needs $O(P)$ images g_j to interpolate G mod each prime. The number of images that Zippel's algorithm and the Hu/Monagan algorithm need to interpolate G depends on the largest coefficient of G. Let $G = \sum_{i=1}^{d_0} a_i(x_1, \ldots, x_n)x_0^i$ where the coefficients a_i are polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$. Let $t = \max(\#a_i)$. Zippel's algorithm needs $O(Dt)$ images. The Hu/Monagan algorithm [7] needs only $O(t)$ images. For the first prime p_1 Hu/Monagan apply a Kronecker substitution to map the GCD computation from $\mathbb{Z}_{p_1}[x_0, x_1, \ldots, x_n]$ into $\mathbb{Z}_{p_1}[x, y]$ then use a modified Ben-Or/Tiwari sparse interpolation to interpolate y.

In Section 2 we present the original Ben-Or/Tiwari interpolation algorithm from [2] and we explain why it must be modified for the GCD problem. We then present the modified Ben-Or/Tiwari interpolation from [7]. In Section 3 we describe our GCD algorithm. It modifies the Hu/Monagan algorithm so that it stops when it has interpolated the smaller of G and one of the cofactors $\bar{A}$ and $\bar{B}$. This leads to a big performance improvement when $\min(\#\bar{A}, \#\bar{B}) \ll \#G$. To further improve the speed of our implementation, we have coded several sub-algorithms in the C language. In particular, we explain how we evaluate the input polynomials A and B which is usually the bottleneck of our algorithm. These implementation details and how we call our C routines from Maple are presented in Section 4. In Section 5 we show how much faster our new GCD algorithm is compared with Maple's implementation of Zippel's algorithm. We end with a short conclusion. We plan to intall our new GCD algorithm into Maple.

[1] If the leading coefficient of G in x_0 is not a constant, they interpolate a multiple of G. See Section 3.2

2 Sparse Polynomial Interpolation

Let $f(x_1,\ldots,x_n) = \sum_{i=1}^{t} a_i M_i(x_1,\ldots,x_n)$ where $t = \#f$ is the number of terms of f and $a_i \in \mathbb{Z}$ and M_i are monomials. Suppose t, a_i and M_i are unknown. The Ben-Or/Tiwari algorithm [2] assumes a term bound $T \geq t$ is given. It interpolates f from the $2T$ values

$$b_j = f(2^j, 3^j, 5^j, \ldots, p_n^j) \quad \text{for} \quad 0 \leq j \leq 2T-1$$

where p_n denotes the n'th prime. Let $m_i = M_i(2,3,5,\ldots,p_n)$ be the monomial evaluations and let $\lambda(z) = \prod_{i=1}^{t}(z - m_i)$.

Algorithm BTinterpolation

Input $b \in \mathbb{Z}^{2T}$ where $b_j = f(2^j, 3^j, 5^j, \ldots, p_n^j)$ for $0 \leq j \leq 2T-1$.
Output $f = \sum_{i=1}^{t} a_i M_i(x_1,\ldots,x_n) \in \mathbb{Z}[x_1,\ldots,x_n]$

1. Compute t and $\lambda(z)$ from b using the Berlekamp-Massey algorithm [10] or using the Euclidean algorithm [1, 22].
2. Factor $\lambda(z) \in \mathbb{Z}[z]$ to get the integer roots m_i.
3. Factor the integers m_i using trial division by $2,3,\ldots,p_n$ to get M_i. E.g. if $m_i = 45000 = 2^3 3^2 5^4$ then $M_i = {x_1}^3 {x_2}^2 {x_3}^4$.
4. Let $V_{ij} = m_i^{j-1}$ for $1 \leq i \leq t, 1 \leq j \leq t$ and let $b = [b_0,\ldots,b_{t-1}]$. Solve the $t \times t$ Vandermonde system $V\,a = b$ for the coefficients a_i.
5. Output $\sum_{i=1}^{t} a_i M_i$.

The evaluations at prime powers ensure that the monomial evaluations $M_i(2,3,\ldots,p_n)$ are distinct. This ensures that the monomials can be uniquely recovered in step 3 and the Vandermonde matrix V in step 4 is non singular. However there is a problem with the prime power choice. When T is not small, the evaluations $b_j = f(2^j, 3^j, \ldots, p_n^j)$ are very large integers. This ruins the efficiency of the algorithm in practice. A staple of Computer Algebra to avoid large intermediate integers is to compute modulo a prime p. The first modification is to evaluate $f(2^j, 3^j, \ldots, p_n^j)$ modulo p and do steps 1,2, and 4 modulo p. Provided $p > m_i$ the monomials will be distinct. If $d_i \geq \deg(f, x_i)$ one may use $m_i \leq 2^{d_1} 3^{d_2} \ldots p_n^{d_n}$ for this purpose. However, such a prime may be too big for us to use machine arithmetic.

In the GCD problem some evaluation points cannot be used.

Definition 2 (Unlucky Evaluation Point)**.** Let $A, B \in \mathbb{Z}_p[x_0, x_1, \ldots, x_n]$, $G = \gcd(A,B)$ and $\bar{A} = A/G$ and $\bar{B} = B/G$. Then $\gcd(\bar{A}, \bar{B}) = 1$. An evaluation point $\alpha \in \mathbb{Z}_p^n$ is *unlucky* if $\deg(\gcd(\bar{A}(x_0,\alpha), \bar{B}(x_0,\alpha)), x_0) > 0$.

Example 2. Let $\bar{A} = x^2 + (y-1)(z-9)x + 3zy$, $\bar{B} = x^2 + 3zy$ and $G = x + y + z$. Here the evaluation points $y = 1$ and $z = 9$ are unlucky as the cofactors are no longer relatively prime at these points. Notice that $y = 1$ and $z = 9$ occur at $j = 0$ and $j = 2$ in the Ben-Or/Tiwari evaluation point sequence $(y = 2^j, z = 3^j)$.

Since we cannot use unlucky evaluation points to interpolate G and any integer could be unlucky, we must use random evaluation points on $[0, p)$ to avoid unlucky evaluation points.

Another problem is the term bound $T \geq t$. If t were known then the Ben-Or/Tiwari algorithm interpolates f using the $2t$ points $(2^j, 3^j, \ldots, p_n^j)$ for $0 \leq j < 2t$. But it is highly unlikely that such information is available. Moreover, good term bounds are not

known. Hu and Monagan adopt the solution of Kaltofen, Lee and Lobo in [11]. For p sufficiently large, if we compute $\lambda(z)$ after $j = 2, 4, 6, \ldots$ points, we will see $\deg(\lambda, z) = 1, 2, \ldots, t-2, t-1, t, t, t, \ldots$ with high probability. Thus we wait until the degree of $\lambda(z)$ does not change.

2.1 Modified Ben-Or/Tiwari Interpolation

Let $A, B \in \mathbb{Z}[x_0, \ldots, x_n]$ and $G = \gcd(A, B)$. Hu and Monagan [7] use an invertible Kronecker substitution $\mathbf{Kr} : \mathbb{Z}[x_0, x_1, \ldots, x_n] \to \mathbb{Z}[x, y]$ to map the GCD computation into one in $\mathbb{Z}[x, y]$. Let $f \in \mathbb{Z}[x_0, x_1, \ldots, x_n]$ and

$$Kr(f) = f(x, y, y^{r_1}, y^{r_1 r_2}, \ldots, y^{r_1 r_2 \cdots r_{n-1}})$$

This mapping is invertible iff $r_i > \deg(f, x_i)$.

Example 3. Let $f(x) = 2x_0^2 x_2 + 3x_0 x_1^3 x_2^2 + 5x_0 x_2^4 + 7x_1 x_2^2$ Here $d_0 = 3, d_1 = 3, d_2 = 4$. Using $r_1 = 4, r_2 = 5$ we have $K_r(f) = f(x, y, y^4) = 2x^2 y^4 + (3y^{11} + 5y^8)x + 7y^9$.

Let $G = \sum_{i=0}^{d_0} c_i(x_1, \ldots, x_n) x_0^i$. Hu and Monagan interpolate the coefficients $Kr(c_i)(y)$ modulo a prime p using the following modified Ben-Or/Tiwari interpolation. The ideas behind it are due to Murao and Fujise [20] and Kaltofen [12].

Let $C(x_1, \ldots, x_n)$ be one of the coefficients and let $f = \mathbf{Kr}(C) = \sum_{i=1}^{t} a_i y^{e_i}$. We first pick a prime $p > e_i$ that is smooth, that is, $p - 1$ has no large prime factors. Next we choose a random generator α for $\mathbb{Z}_p$. Now for some $L > 0$ we compute the values $f(\alpha^j)$ for $1 \le j \le 2L$. The following algorithm interpolates f for $L \ge t + 1$. Here the monomial evaluations $m_i = \alpha^{e_i}$ and $\lambda(z) = \prod_{i=1}^{t}(z - m_i)$.

Algorithm ModifiedBT
Input $b \in \mathbb{Z}_p^{2L}$ where $b_j = f(\alpha^j)$ for $1 \le j \le 2L$ and α a generator for $\mathbb{Z}_p^*$.
Output $f = \sum_{i=1}^{t} a_i y^{e_i} \in \mathbb{Z}_p[y]$ with $t < L$.

1. Compute $\lambda(z)$ from b_j using the Berlekamp-Massey algorithm or the Euclidean algorithm and let $t = \deg(\lambda, z)$.
   If $t = L$ **return** FAIL. // $t < L$ means $\lambda(z)$ is correct with high probability [11]
2. Factor $\lambda(z)$ over $\mathbb{Z}_p$ to get the roots m_i.
   If $\#m \ne t$ **return** FAIL. // as $\lambda(z)$ is incorrect
3. Solve $\alpha^{e_i} = m_i$ in $\mathbb{Z}_p$ for e_i with $0 \le e_i < p - 1$.
4. Let $W_{ij} = m_i^j$ for $1 \le i \le t, 1 \le j \le t$ and let $b = [b_1, \ldots, b_t]$.
   Solve the $t \times t$ shifted Vandermonde system $W\, a = b$ for the coefficients a_i.
5. Output $\sum_{i=1}^{t} a_i y^{e_i}$.

The Berlekamp-Massey algorithm and the Euclidean algorithm both do $O(L^2)$ arithmetic operations in $\mathbb{Z}_p$. The Cantor-Zassenhaus algorithm from [4] can be used to do the factorization in step 2. Maple uses this algorithm. The Maple implementation does $O(t^2 \log p)$ arithmetic operations in $\mathbb{Z}_p$. Usually step 2 dominates the cost.

Step 3 is a discrete logarithm problem. We need to compute $\log_\alpha m_i$ in the finite field $\mathbb{Z}_p$. No polynomial time algorithm is known for this problem in general. Let $p - 1 = \prod_{i=1}^{k} p_i$ be prime factorization of $p - 1$. The discrete logarithm can be computed using

$O(\sum_{i=1}^{k}(\log p + \sqrt{p_i}))$ arithmetic operations using the Pohlig/Hellman algorithm [21]. Thus if $p-1$ has only small prime factors p_i then computing discrete logarithms is feasible. The numtheory[mlog](m,alpha,p) command in Maple computes $\log_\alpha m$ in $\mathbb{Z}_p$.

Step 4 is a shifted transposed Vandermonde system. Let

$$W = \begin{bmatrix} m_1 & m_2 & \dots & m_t \\ {m_1}^2 & {m_2}^2 & \dots & {m_t}^2 \\ \vdots & \vdots & \vdots & \vdots \\ {m_1}^t & {m_2}^t & \dots & m_t^t \end{bmatrix} \text{ and } V = \begin{bmatrix} m_1^0 & m_2^0 & \dots & m_t^0 \\ {m_1}^1 & {m_2}^1 & \dots & {m_t}^1 \\ \vdots & \vdots & \vdots & \vdots \\ {m_1}^{t-1} & {m_2}^{t-1} & \dots & m_t^{t-1} \end{bmatrix}.$$

Observe that $W = VD$ where D is the diagonal matrix with $D_{ii} = m_i$ for $1 \le i \le t$. To solve $Wa = b$ we first solve $Vc = b$ for c. The linear system $Vc = b$ is a transposed Vandermonde system. In [25] Zippel solves it in $O(t^2)$ arithmetic operations in $\mathbb{Z}_p$ and space for $O(t)$ elements of $\mathbb{Z}_p$. Now $W = VD$ and $Wa = b$ implies $V(Da) = b$ so $c = Da$. Hence $c_i = m_i a_i$ so $a_i = m_i^{-1} c_i$. We can compute a from c using t multiplications and t inverses.

Hu and Monagan uses an alternative way to randomize the evaluations. They pick a generator α for $\mathbb{Z}_p$ and a random shift s from $[1, p-2]$ and evaluate at $y = \alpha^j$ for $j = s, s+1, \dots, s+2L-1$. Then $W_{ij} = m_i^{s+j}$, $D_{ii} = m_i^s$ and one obtains $a_i = m_i^{-s} c_i$.

We return to the problem of the size of the prime p needed in in the Ben-Or/Tiwari interpolation algorithm. Let $n = 8$ and $d_i = \deg(f, x_i) = 10$. Algorithm BTinterpolation requires $p > 2^{10}3^{10}\dots 19^{10} = 7.4 \times 10^{69}$ in general. Such a prime is too big for a 64 bit computer. Algorithm ModifiedBT requires $p > e_i$. Since $e_i < r_1 r_2 \dots r_n$ where $r_i = d_i + 1$, we need $p > 11^8 = 214,358,881$. Our Maple code has a table of 62, 128, 256, 512, and 1024 bit smooth primes. The first 62 bit prime in our table is the smooth prime

$$p = 4,601,552,919,265,804,289 = 61 \times 67 \times 2^{50} + 1 = 2^{61.99}.$$

3 The GCD Algorithm

As in Zippel's GCD algorithm, the Hu/Monagan GCD algorithm first computes and removes the content from the input polynomials A and B.

Definition 3 (Content)**.** Let $f \in \mathbb{Z}[x_0, x_1, ..., x_n]$, $d = \deg(f, x_0$ and $f = \sum_{i=0}^{d} a_i(x_1, \dots, x_n)x_0^i$ where $a_i \in \mathbb{Z}[x_1, \dots, x_n]$. Define $\text{cont}(f, x_0) = \gcd(a_0, a_1, \dots, a_d)$ to be the content of f. If $\text{cont}(f, x_0) = 1$ we say f is *primitive*.

Example 4. If $f = (6x_1^3 - 6)x_0 + (9x_1^2 - 9)$ then $\text{cont}(f, x_0) = \gcd(6x_1^3 - 6, 9x_1^2 - 9) = 3x_1 - 3$.

The GCD computations that occur in the contents are in $\mathbb{Z}[x_1, \dots, x_n]$, that is, in one less variable, so they can be done recursively. Note, the Maple command content(f,x_0) computes the content. We now give an overview of our GCD algorithm which handles the content of the GCD.

Algorithm GCD
Input non-zero polynomials $A, B \in \mathbb{Z}[x_0, x_1, \dots, x_n]$.
Output $G = \gcd(A, B)$.
1: $ca, cb \leftarrow \text{cont}(A, x_0), \text{cont}(B, x_0)$.
2: $A, B \leftarrow A/ca, B/cb$. // Now A and B are primitive.
3: **repeat**

4: $H \leftarrow \mathsf{MGCD}(A, B)$
5: **until** $H \neq FAIL$ and $H|A$ and $H|B$.
6: **return** $H \times \gcd(ca, cb)$.

Algorithm MGCD computes a multiple H of G. See Section 3.2. Algorithm MGCD first calls algorithm PGCD which chooses a Kronecker substitution Kr, then the first prime p_1, computes $Kr(H_1) = Kr(H) \bmod p_1$ then inverts Kr to obtain H_1. Then MGCD chooses primes $p_2, p_3, \ldots$, computes $A_k = A \bmod p_k$, $B_k = B \bmod p_k$ and calls Zippel's algorithm SGCD with inputs A_k, B_k, H_1 and computes $H_k = H \bmod p_k$. SGCD assumes $\text{supp}(H_1) = \text{supp}(H)$, that is, SGCD assumes the monomials in H_1 and in H are the same. This will not be the case if p_1 divides any integer coefficient of H. After computing $H_1, H_2, H_3, \ldots$ algorithm MGCD applies the Chinese remainder theorem to recover the integer coefficients of H. Finally it divides H by $\text{cont}(H, x_0)$ to get G.

During the algorithm several types of failure can occur. The basic strategy is to choose primes and evaluation points so that they occur with low probability and detect the failures. If a failure is detected, we restart the algorithm with a new Kronecker substitution and new primes $p_1, p_2, \ldots$. We give an example of a failure that goes undetected in MGCD.

Example 5. Let $G = 1$, $\bar{A} = x_0^2 + p_1 p_2 x_0 + x_1 x_2$ and $\bar{B} = x_0^2 + x_1 x_2$ where p_1 is the prime chosen by PGCD and p_2 is the first prime chosen by MGCD. PGCD will compute $H_1 = G\bar{B} = B \bmod p_1$ and SGCD will compute $H_2 = G\bar{B} = B \bmod p_2$. After Chinese remaindering MGCD will have $H = B$. Then since $\text{cont}(B, x_0) = 1$ MGCD outputs B not G.

All cases of undetected failure are caught in Algorithm GCD by the trial divisions $H|A$ and $H|B$. Algorithm PGCD guarantees $\deg(H_1, x_0) \geq \deg(G, x_0)$. Since the inputs A and B to MGCD are primitive and the output H of MGCD is primitive, if $H|A$ and $H|B$ over $\mathbb{Z}$, these conditions imply $H = \pm G$.

Let us focus on the key algorithm PGCD where most of the work occurs. PGCD chooses a Kronecker substitution Kr and a smooth prime p to map the computation of $\gcd(A, B)$ into $\mathbb{Z}_p[x, y]$. Then PGCD chooses a random generator α for $\mathbb{Z}_p^*$ and computes

$$g_j = \gcd(Kr(A)(x, \alpha^j), Kr(B)(x, \alpha^j)) \quad \text{for } j = 1, 2, \ldots$$

using the Euclidean algorithm for GCD computations in $\mathbb{Z}_p[x]$. It is possible that α^j is unlucky, that is, $\deg(g_j, x) > \deg(Kr(G), x)$. If any α^j is unlucky we cannot interpolate $Kr(G)$ using algorithm ModifiedBT. Also, in $\mathbb{Z}_p[x]$, because $\mathbb{Z}_p$ is a field, these univariate images g_j are unique up to a non-zero scalar in $\mathbb{Z}_p$. To interpolate $Kr(G)$ using polynomial interpolation we need $\text{LC}(g_j, x) = \text{LC}(Kr(G), x)(\alpha^j)$. But we do not know $\text{LC}(Kr(G), x)$ because we do not know G! This is called the leading coefficient problem.

3.1 Detecting Unlucky Evaluation Points

In [3] Brown applies Theorems 1 and 2 below to detect unlucky evaluation points in $\mathbb{Z}_p$.

Definition 4 (Leading Coefficient)**.** Let $f = \sum_{i=0}^{d} a_i(y) x^i$ be a polynomial in $R[x, y]$ with $a_d \neq 0$. The leading coefficient of f, denoted $\text{LC}(f, x)$, is $a_d(y)$.

Theorem 1. Let $A, B \in \mathbb{Z}_p[x, y]$ and $\alpha \in \mathbb{Z}_p$. Let $G = \gcd(A, B)$ and $g = \gcd(A(x, \alpha), B(x, \alpha))$. If $\text{LC}(A, x)(\alpha) \neq 0$ then (1) $\deg(g, x) \geq \deg(G, x)$ and (2) $g|G(x, \alpha)$.

Theorem 2. Let A, B be non-zero polynomials in $\mathbb{Z}_p[x, y]$ and $\alpha \in \mathbb{Z}_p$. Let $R = \text{res}(A, B, x)$ be the Sylvester resultant of A and B. Then (1) $R \in \mathbb{Z}_p[y]$ and (2) α is unlucky $\Longrightarrow R(\alpha) = 0$.

For a proof of Theorem 1 see Lemma 7.3 in [5]. Theorem 1 says, provided α^j satisfies $\text{LC}(Kr(A), x)(\alpha^j) \neq 0$ then g_j satisfies either $\deg(g_j, x) > \deg(Kr(G), x)$ or $g_j = Kr(G)(x, \alpha^j)$ upto multiplication by a unit in $\mathbb{Z}_p$. Brown's strategy is to use the g_j of least degree to interpolate $Kr(G)$. Since we need all g_j to be lucky for Algorithm ModifiedT we stop if any α^j is unlucky or $\text{LC}(Kr(A), x)(\alpha^j) = 0$ and restart MGCD.

For a proof of Theorem 2 see Lemma 4 in [8]. Theorem 2 tells us that the number of unlucky evaluation points is finite since $\deg(R, y) \leq \deg(A)\deg(B)$ by the Bezout bound. Provided we choose $p \gg \deg(A)\deg(B)$ then most evaluation points are lucky and we will detect any unlucky evaluation point with high probability. It is possible, however, that all α_j are unlucky in which case we interpolate a polynomial H with $\deg(H, x_0) > \deg(G, x_0)$.

It is possible that Kr is unlucky, that is, $\deg(\gcd(Kr(A), Kr(B)), x) > \deg(G, x_0)$. It is also possible that a prime p is unlucky, that is, $\deg(\gcd(A \bmod p, B \bmod p), x_0) > \deg(G, x_0)$. Our design of algorithm MGCD will detect most failures.

3.2 Leading Coefficient Correction

Let $g_j = \gcd(Kr(A)(x, \alpha^j), Kr(B)(x, \alpha^j))$ in $\mathbb{Z}_p[x]$. Suppose $\deg(g_j, x) = \deg(G, x_0)$. Then by Theorem 1 g_j equals $Kr(G)(x, \alpha^j) \bmod p$ up to multiplication by a unit in $\mathbb{Z}_p$. For a GCD computation in $\mathbb{Z}_p[x]$ we normally output $\text{monic}(g_j)$, the monic associate of g_j. Thus

$$g_j = Kr(G)(x, \alpha^j)/\text{LC}(Kr(G), x)(\alpha_j) = \left[\frac{Kr(G)}{\text{LC}(Kr(G), x)}\right](\alpha^j).$$

Thus if $\text{LC}(Kr(G), x)$ is a non-constant polynomial in y then the images g_j are images of a rational function in $\mathbb{Z}_p(y)[x]$ and we cannot use polynomial interpolation to interpolate y. The solution adopted by Zippel [24] and Hu/Monagan [7] is to use the fact that

$$A = G\bar{A} \implies \text{LC}(Kr(A), x) = \text{LC}(Kr(G), x) \times \text{LC}(Kr(\bar{A}), x)$$

and multiply g_j by $\text{LC}(Kr(A), x)(\alpha^j)$ so that we interpolate the polynomial $Kr(H) = \text{LC}(Kr(\bar{A}), x) \times Kr(G)$ which is a multiple of $Kr(G)$. The polynomial H can have more terms than G.

Example 6. Let $G = x_1x_0^2 + (x_1^2x_2 + 3)x_0 + 3x_2x_1$ $\bar{A} = (x_1 + x_2)x_0 + 3x_1x_2$. We have $\text{LC}(G, x_0) = x_1, \text{LC}(\bar{A}, x_0) = x_1 + x_2$, $H = (x_1 + x_2)G$, $\#G = 4$ and $\#H = 8$.

In MGCD, after Chinese remaindering we have H. To recover G from H we compute $\text{cont}(H, x_0) = \pm\text{LC}(\bar{A}, x_0)$ and output $H/\text{cont}(H, x_0) = \pm G$. Note, we could multiply g_j by $\text{LC}(Kr(B), x)(\alpha^j)$ instead if $\text{LC}(Kr(B), x)$ has fewer terms than $\text{LC}(Kr(A), x)$.

3.3 Interpolating G and $\bar{A}$ simultaneously

In our experiments we observed that with $\#A$ and $\#B$ fixed, as $\#G$ increases, the time for our GCD algorithm increases – see column MGCD1 in Table 1. This is simply because we need more images g_j to interpolate H. We noticed that when $\#G \gg \bar{A}$ that Maple's factorization command factor was faster than our new GCD code – see column factor in Table 1. Why bother with a complicated GCD algorithm if we can simply factor A and B

to get G instead? This motivated us to redesign our GCD algorithm to try to interpolate the smaller of $G, \bar{A}, \bar{B}$. Interchanging the inputs A and B if necessary, let us assume $\#A \leq \#B$. Let $H = \mathrm{LC}(\bar{A}, x_0)G$ and $C = \mathrm{LC}(G, x_0)\bar{A}$. Let

$$Kr(H) = \sum_{i=0}^{\deg(Kr(H),x)} h_i(y)x^i \text{ and } Kr(C) = \sum_{i=0}^{\deg(Kr(C),x)} c_i(y)x^i.$$

The number of images g_j needed to interpolate the smaller of $Kr(H)$ and $Kr(C)$ depends on $t = \min(\max \#h_i, \max \#c_i)$.

Algorithm PGCD
Input $A, B, \in \mathbb{Z}[x_0, x_1, \ldots, x_n]$.
Output (false, H, p) or (true, C, p) or FAIL.
Assert $\mathrm{LC}(H, x_0) = \mathrm{LC}(A, x_0) \bmod p$, $\mathrm{LC}(C, x_0) = \mathrm{LC}(A, x_0) \bmod p$.

1 Pick a new Kronecker substitution Kr with $r_i \geq \deg(A, x_i)$ for $1 \leq i \leq n$.
2 Pick a new smooth prime $p > \prod_{i=1}^{n} r_i$.
3 Pick a generator α for $\mathbb{Z}_p$ at random.
4 $A_p, B_p \leftarrow A \bmod p, B \bmod p$.
5 **for** $j = 1, 2, 3, \ldots$ **do**
6 $a_j, b_j \leftarrow Kr(A_p)(x, \alpha^j), Kr(B_p)(x, \alpha^j)$.
7 **if** $\deg(a_j, x) < \deg(A, x_0)$ **return** FAIL.
8 $g_j \leftarrow \mathrm{monic}(\gcd(a_j, b_j))$
9 **if** $j > 1$ and $\deg(g_j, x) \neq \deg(g_1, x)$ **return** FAIL.
10 $d_j \leftarrow a_j / g_j$.
11 $g_j \leftarrow \mathrm{LC}(a_j, x) \times g_j$
12 **if** $j \in \{4, 6, 8, 10, \ldots\}$ // Try to interpolate $Kr(H)$ or $Kr(C)$
13 $f \leftarrow$ SparseInterpolate$([g_i : 1 \leq i \leq j], \alpha)$.
14 **if** $f \neq FAIL$ **return** (false, $Kr^{-1}(f), p$).
15 $f \leftarrow$ SparseInterpolate$([d_i : 1 \leq i \leq j], \alpha)$.
16 **if** $f \neq FAIL$ **return** (true, $Kr^{-1}(f), p$).
17 **end if**
18 **end for**

The requirement $r_i > \deg(A, x_i)$ guarantees we can invert $Kr(H)$ and $Kr(C)$. Lines 7 and 9 apply Theorem 1 to detect unlucky α^j. To interpolate $Kr(H)$ from g_j and $Kr(C)$ from d_j simultaneously we could try the Algorithm ModifiedBT for $j \in \{4, 6, 8, 10, \ldots\}$ until we succeed. We use it to interpolate each $h_i(y)$ using coeff(g_j, x^i) and each $c_i(y)$ using coeff(d_j, x^i). Algorithm PGCD will stop when $j > 2t$ where $t = \min(\max(\#h_i), \max(\#c_i))$. In practice, we found this strategy does too much work in step 1 of Algorithm ModifiedBT. So we try when $j \in \{4, 6, 10, 16, \ldots, 2F_n, \ldots\}$ instead.

Subroutine SparseInterpolate is presented below. It applies Algorithm ModifiedBT to interpolate each $h_i(y)$ using the coeff(g_j, x^i) then each $c_i(y)$ using the coeff(d_j, x^i).

Subroutine SparseInterpolate
Input $g_1, g_2, \ldots, g_{2L} \in \mathbb{Z}_p[x]$ and $\alpha \in \mathbb{Z}_p$.
Output FAIL or $f \in \mathbb{Z}_p[x, y]$.
Assert $f(x, \alpha^j) = g_j$ for $1 \leq j \leq 2L$ and $\#\mathrm{coeff}(f, x^i) < L$ for $0 \leq i \leq d = \deg(f, x)$.

1 $d \leftarrow \deg(g_1, x)$
2 **for** $i = 0, 1, \ldots, d$ **do**
3 $\quad b \leftarrow [\mathsf{coeff}(g_j, x^i) : 1 \le j \le 2L]$
4 $\quad f_i \leftarrow \mathsf{ModifiedBT}(b, \alpha)$
5 $\quad$ **if** $f_i =$ FAIL **return** FAIL. // else $\#f_i < L$.
6 **end for**
7 **return** $\sum_{i=0}^{d} f_i x^i$.

3.4 Algorithms SGCD and MGCD

Algorithm SGCD applies Zippel's sparse interpolation to interpolate $H \bmod p$ using the support of H_1.

Algorithm SGCD
Input $A, B, H_1 \in \mathbb{Z}_p[x_0, x_1, \ldots, x_n]$.
Output H with $\mathrm{LC}(H, x_0) = \mathrm{LC}(A, x_0)$ or FAIL.

1 $d \leftarrow \deg(H_1, x_0)$.
2 Let $H_1 = \sum_{i=0}^{d} \sum_{j=1}^{t_i} h_{ij} M_{ij}(x_1, \ldots, x_n) x_0^i$ where $h_{ij} \in \mathbb{Z}_p \backslash \{0\}$.
3 Pick β from $\mathbb{Z}_p^n$ at random.
4 **for** $i = 0, 1, 2, \ldots, d$ **do**
5 $\quad m_i \leftarrow \{M_{ij}(\beta) \bmod p : 1 \le j \le t_i\}$.
6 $\quad$ **if** $|m_i| < t_i$ **return** FAIL.
7 **end for**
8 Set $t \leftarrow \max t_i$.
9 **for** $j = 1, 2, 3, \ldots, t$ **do**
10 $\quad a_j, b_j \leftarrow A(x_0, \beta^j), B(x_0, \beta^j)$.
11 $\quad$ **if** $\deg(a_j, x_0) \neq \deg(A, x_0)$ **return** FAIL.
12 $\quad g_j \leftarrow \mathrm{monic}(\gcd(a_j, b_j))$.
13 $\quad$ **if** $\deg(g_j, x_0) \neq d$ **return** FAIL.
14 $\quad g_j \leftarrow \mathrm{LC}(a_j, x_0) \cdot g_j$.
15 **end for**
16 **for** $i = 0, 1, 2, \ldots, d$ **do**
17 $\quad W_{kj} \leftarrow m_{ik}^j$ for $1 \le k, j \le t_i$.
18 $\quad b \leftarrow [\mathsf{coeff}(g_j, x_0^i) : 1 \le j \le t_i]$.
19 $\quad$ Solve the shifted Vandermonde system $W a_i = b$ for $a_i \in \mathbb{Z}_p^{t_i}$.
20 **end for**
21 **return** $H = \sum_{i=0}^{d} \sum_{j=1}^{t_i} a_{ij} M_{ij} x^i$.

Algorithm SCOF interpolates $C \bmod p$ using the support of C_1. It is the same as SGCD with line 13 replaced with $c_j \leftarrow a_j / g_j$, and line 17 using c_j instead of g_j.

Line 5 of Algorithm SGCD requires the monomial evaluations in each set m_i to be distinct so that in line 16 $\det(W) \neq 0$. This is a birthday problem: $|m_i| = t_i$ with high probability if $p \gg t_i^2$.

Repeated use of Algorithm SGCD will identify if $\deg(H_1, x_0) > \deg(G, x_0)$ thus identifying unlucky Kr, p_1, or α^j. Similarly for SCOF. It is also possible that $\mathrm{supp}(H_1) \neq \mathrm{supp}(H)$. For example, if p_1 divides any integer coefficient of H. In MGCD, after each call to SGCD, we use an additional evaluation point to check if $\mathrm{supp}(H_1)$ is correct. We present MGCD.

Algorithm MGCD
Input $A, B \in \mathbb{Z}[x_0, x_1, \ldots, x_n]$.
Output G or FAIL.

2 $cof, H_1, p_1 \leftarrow \mathsf{PGCD}(A, B)$.
3 $H, M \leftarrow H_1, p_1$.
4 **for** $k = 2, 3, \ldots$ **do**
5 $H' \leftarrow H$
6 Pick a new prime p_k such that $\mathrm{LC}(A, x_0) \bmod p_k \neq 0$.
7 $A_k, B_k \leftarrow A \bmod p_k, B \bmod p_k$.
8 $H_k \leftarrow$ **if** cof **then** $\mathsf{SCOF}(A_k, B_k, H_1)$ **else** $\mathsf{SGCD}(A_k, B_k, H_1)$ **end if**
9 **if** $H_k = FAIL$ **return** FAIL.
10 **if** cof and not $\mathsf{CheckCof}(A_k, B_k, H_k)$ **return** FAIL.
11 **if** not cof and not $\mathsf{CheckGcd}(A_k, B_k, H_k)$ **return** FAIL.
12 $\Delta \leftarrow M^{-1}(H_k - H) \bmod p_k$, $H \leftarrow H + \Delta M$, $M \leftarrow p_k M$.
13 $H \leftarrow \mathsf{mods}(H, M)$. // use the symmetric range for $\mathbb{Z}_M$
14 **if** $H = H'$ **then**
15 $H \leftarrow H/\mathrm{cont}(H, x_0)$.
16 **if** not cof **return** H.
17 if $H|A$ **return** A/H **else return** FAIL **end if**
18 **end if**
19 **end for**

Subroutine CheckGcd
Input $A_k, B_k, H_k \in \mathbb{Z}_p[x_0, x_1, \ldots, x_n]$ with $\mathrm{LC}(A_k, x_0) = \mathrm{LC}(H_k, x_0)$.
Output false $\implies H_k \neq \mathrm{LC}(\bar{A}, x_0)G \bmod p$.

1 Pick $\beta \in \mathbb{Z}_p$ at random **until** $\mathrm{LC}(A_k, x_0)(\beta) \neq 0$.
2 Set $a, b = A_k(x, \beta), B_k(x, \beta)$, $g \leftarrow \gcd(a, b)$ and $h \leftarrow H_k(x, \beta)$.
3 **if** $g|h$ and $h|g$ in $\mathbb{Z}_p[x]$ **return** true **return** false.

Subroutine CheckCof
Input $A_k, B_k, H_k \in \mathbb{Z}_p[x_0, x_1, \ldots, x_n]$ with $\mathrm{LC}(A_k, x_0) = \mathrm{LC}(H_k, x_0)$.
Output false $\implies H_k \neq \mathrm{LC}(G, x_0)\bar{A} \bmod p$.

1 Pick $\beta \in \mathbb{Z}_p$ at random **until** $\mathrm{LC}(A_k, x_0)(\beta) \neq 0$.
2 Set $a, b = A_k(x, \beta), B_k(x, \beta)$, $g \leftarrow \gcd(a, b)$, $c = a/g$ and $h \leftarrow H_k(x, \beta)$.
3 **if** $c|h$ and $h|c$ in $\mathbb{Z}_p[x]$ **return** true **return** false.

All failures that go undetected in MGCD are caught by the trial divisions $H|A$ and $H|B$ in Algorithm GCD. When any failure is detected we restart MGCD which chooses a new Kronecker substitution Kr and new smooth prime p_1. Although it is possible to recover from some failures, doing so complicates the code. For example, in Algorithm SGCD the reader may have asked, if $|m_i| < t_i$, why don't you simply choose a new $\beta \in \mathbb{Z}_p^n$ and try again. Indeed one could do that. But each attempt to recover from a failure complicates the code. We think it is better to keep the code simple and choose large primes in Algorithms PGCD and SGCD so that the probability of failure is very low. For this reason we use 62 bit primes or larger as needed in PGCD and 62 bit primes in SGCD.

4 Implementation Notes

We've coded the following routines in C to speed up our implementation. Our C code supports primes $p < 2^{63}$. The complexities of our subroutines are given in blue. They give the number of arithmetic operations in $\mathbb{Z}_p$ done by those subroutines.

1 Evaluations $Kr(A)(x, \alpha^j)$ and $Kr(B)(x, \alpha^j)$ ..... $O(D + ns + st)$
2 Finding $\lambda(z)$ (step 1 of Alg. ModifiedBT) ..... $O(t^2)$
3 Factoring $\lambda(z)$ (step 2 of Alg. ModifiedBT) ..... $O(t^2 \log p)$
4 Solving $t \times t$ shifted Vandermonde systems ..... $O(t^2)$

Here $D = \sum_{i=1}^{n} \max(\deg(A, x_i), \deg(B, x_i))$, $s = \#A + \#B$ is the number of terms in the input, and t is the number of terms in the largest polynomial we interpolate. Recall that we interpolate $Kr(H) = \mathrm{LC}(Kr(\bar{A}), x)\, Kr(G)$ and $Kr(C) = \mathrm{LC}(Kr(G), x)\, Kr(\bar{A})$ simultaneously. Let $Kr(H) = \sum_{i=0} h_i(y)x^i$ and $Kr(C) = \sum_{i=0} c_i(y)x^i$ for h_i and c_i in $\mathbb{Z}[y]$. We must interpolate either all h_i or all c_i. Thus $t = \min(\max(\#h_i), \max(\#c_i))$. Note, in Table 1 the values of t are given for each experiment.

4.1 Maple's data representation for polynomials

We digress to detail how Maple represents polynomials because we want to evaluate them using C code. Maple uses two data structures for polynomials, the SUM-OF-PROD data structure and the POLY data structure. POLY was added to Maple in 2013 by Monagan and Pearce [14, 15] to speed up polynomial arithmetic. Figure 1 shows the POLY data structure for the polynomial $f = 9\,xy^3z - 4\,y^3z^2 - 6\,xy^2z - 8\,x^3 - 5$. If $M = x_1^{d_1} x_2^{d_2} \ldots x_n^{d_n}$

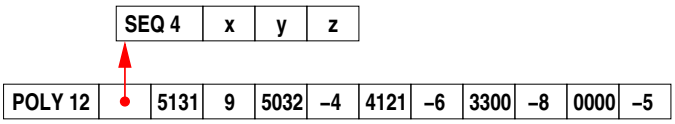

Figure 1: Maple's representation for $f = 9\,xy^3z - 4\,y^3z^2 - 6\,xy^2z - 8\,x^3 - 5$.

is a monomial in f, and $d = d_1 + d_2 + \ldots d_n$ is the degree of M then M is encoded as the integer

$$dB^n + d_1B^{n-1} + \cdots + d_n \quad \text{where} \quad B = \lfloor 64/(n+1) \rfloor.$$

For example, in Figure 1 the first monomial xy^3z with total degree 5 is encoded as the integer $5 \cdot 2^{48} + 2^{32} + 3 \cdot 2^{16} + 1$. This is depicted as 5131 in Figure 1.

When does Maple use POLY instead of SUM-OF-PROD? If a polynomial f has (i) all integer coefficients, (ii) more than one term, (iii) is not linear, and (iv) all monomials in f can be encoded in a 64 bit integer using B bits for d_i and $64 - nB$ bits for d, then it is encoded using POLY otherwise the SUM-OF-PROD data structure is used.

What is the advantage of POLY? One can compare monomials in POLY using a 64 bit integer comparison and one can multiply monomials in POLY using a 64 bit integer addition. This is much faster than comparing and multiplying two monomials in the PROD representation. Also, the monomial representation is more compact. The monomial xy^2z^3 in POLY uses one 64 bit word whereas in the SUM-OF-PROD representation it uses 7 words.

4.2 Evaluating sparse polynomials

The most expensive part of our GCD algorithm is usually the evaluations of $Kr(A)(x,y)$ and $Kr(B)(x,y)$ at $y = \alpha^j$. This is because we interpolate the smaller of H and C it will almost always be the case that $t \ll \#A + \#B$, that is, what we interpolate (H or C) is much smaller than the size of the input polynomials A and B. How can we speed up the evaluations $Kr(A)(x,\alpha^j)$ and $Kr(B)(x,\alpha^j)$?

We first note that we do not need to explicitly apply the Kronecker map Kr. Instead, given $A(x_0,x_1,\ldots,x_n)$, with $Kr(A) = A(x,y,y^{r_1},\ldots,y^{r_1 r_2 \cdots r_{n-1}})$ we compute

$$\beta_1 = \alpha,\ \beta_2 = \beta_1^{r_1},\ldots,\ \beta_n = \beta_{n-1}^{r_{n-1}}$$

and then we use

$$Kr(A)(x,\alpha^j) = A(x,\beta_1^j,\beta_2^j,\ldots,\beta_n^j).$$

We could use Maple's Eval command to compute $A(x,\beta_1^j,\ldots,\beta_n^j) \bmod p$. Although Eval is coded in C, because Maple would do each evaluation independently, it cannot take advantage of the geometric point sequence $\beta_1^j,\beta_2^j,\ldots,\beta_n^j$. Suppose $A = \sum_{i=1}^{s} a_i x_0^{e_i} M_i$ where $s = \#A$ and M_i are monomials in $x_1,\ldots,x_n$. If we pre-compute the monomial evaluations $m_i = M_i(\beta_1,\ldots,\beta_n)$, then we can exploit

$$M_i(\beta_1^j,\ldots,\beta_n^j) = M_i(\beta_1,\ldots,\beta_n)^j = m_i^j$$

to speed up evaluation of A by a factor of n as follows. Initializing $C = [c_1,\ldots,c_s]$ we compute

1 **for** $j = 1,2,\ldots,t$ **do**
2 $\quad C_i \leftarrow C_i \times m_i$ for $1 \le i \le s$.
3 $\quad g \leftarrow \sum_{i=1}^{s} C_i x^{e_i}$. // $g = Kr(A)(x,\alpha^j)$.
4 **end for**

This algorithm does st multiplications in step 2 plus some additions in step 3 plus the work to compute the monomial evaluations m_i. Our first implementation was the following Maple code. The coeffs command creates outputs the sequence $a_1,\ldots,a_s$ and assigns M the sequence $M_1,\ldots,M_s$.

```
> a := coeffs(A,indets(A),'M');
> a := [a]; M := [M]; s := nops(a);
> C := Array(1..s,a,datatype=integer[8]);
> m := Eval(M,{seq(x[i]=beta[i],i=1..n)}) mod p;
> e := [seq( degree(m[i],x[0]), i=1..s )];
> e := Array(1..s,e,datatype=integer[4]);
> m := Array(1..s,subs(x[0]=1,m),datatype=integer[8]);
```

Now, for $j = 1,2,\ldots,t$ we call two C programs from Maple. The first has inputs C, m and the prime p. It computes $C_i \leftarrow C_i \times m_i$ for $1 \le i \le s$. The second has inputs C, e, g and p. It assembles the polynomial $\sum_{i=1}^{s} C_i x^{e_i}$ in the array g.

It turned out that the setup cost was expensive. In particular for large benchmarks where our input polynomials A and B have a million terms or more, creating and evaluating the monomials $M_i(x_1,\ldots,x_n)$ in Maple is expensive. Why? If A is stored in the POLY representation, the Maple command coeffs(A,indets(A),'M'); creates a sequence of monomials in M in the SUM-OF-PROD representation. For example, in Figure 1 the monomial xy^3z which is encoded as the integer $5 \cdot 2^{48} + 2^{32} + 3 \cdot 2^{16} + 1$ is extracted as the 7 word array

PROD 7	x	1	y	3	z	1

We have implemented a C subroutine getsupp64s$(A, s, \beta, n, x, d, C, e, m, p)$ that on input of the arrays A and β computes the arrays C, e and m. Note, the input x allows us to use any of the variables x_i as the main variable (instead of x_0) and the input $d = \deg(A, x)$. Subroutine getsupport64s constructs arrays of the powers $[1, \beta_i, \ldots, \beta_i^{d_i}]$ for $1 \le i \le n$ where $d_i = \deg(f, x_i)$ then uses these to compute the monomial evaluation m_i. It then uses C bit operators to unpack the monomial encodings.

Note, Maple's C interface does not directly allow us to pass a Maple polynomial to a C subroutine. To circumvent this we pass a pointer to the Maple polynomial to our C routine getsupp64s. The following Maple code, courtesy of Paul DeMarco of Maplesoft, computes a pointer to a Maple object f.

```
> MapleMaxint := 2^63-1;
> pointer := addressof(f)-4*MapleMaxint;
```

If A and B are represented as POLYs, we use getsupport64s to compute the arrays C, e, and m directly. Otherwise we use the Maple code to compute C, e and m.

We mention one further important efficiency consideration. For the multiplications in $\mathbb{Z}_p$ needed to compute the monomial evaluations m_i and to compute $C_i = m_i \times C_i$, for a 63 bit prime p, we do not use the hardware division instruction because it is very expensive. Instead we use Roman Pearce's implementation of the Möller/Granlund algorithm [13] which multiplies by an integer inverse of p instead.

5 Timing Benchmarks

We have benchmarked our code on an 8 core Intel Xeon E5 2660 processor with 64 gigabytes of RAM. In Table 5 we compare Maple's GCD algorithm (column gcd) with our new GCD algorithm (columns MGCD1 and MGCD2). The input polynomials $A = G\bar{A}$ and $B = G\bar{B}$ were created by first creating $G, \bar{A}, \bar{B}$ in $\mathbb{Z}[x_1, \ldots, x_8]$ with $\#G$, $\#\bar{A}$, $\#\bar{B}$ non-zero terms then multiplying $G \times \bar{A}$ and $G \times \bar{B}$. Each term in $G, \bar{A}, \bar{B}$ was created randomly. The monomials $M = x_1^{e_1} x_2^{e_2} \cdots x_8^{e_8}$ were chosen uniformly at random from those with total degree at most 30 and the integer coefficients were chosen uniformly at random from $[0, 2^{100})$. To create such a polynomial with T terms we used Maple's randpoly command as follows.

```
> X := [x1,x2,x3,x4,x5,x6,x7,x8];
> C := rand(2^100);
> f := randpoly(X,degree=30,terms=T,coeffs=C);
```

Table 5 has four sets of timings. In each set of timings the number of terms in G increases by a factor of 10 and the number of terms of the cofactors $\bar{A}$ and $\bar{B}$ decreases by a factor of 10 so that the number of terms of the inputs A and B is held constant.

The time reported in column gcd is for Maple's gcd command. Maple is using Zippel's algorithm [24]. The Maple implementation of Zippel's algorithm is described in [9]. The time reported in column factor is the time for Maple's factor command to factor both A and B. The factor code Maple has been using since Maple2019 is the work of Monagan and Tuncer [16, 17, 18].

The timings in column MGCD1 are for our Maple implementation of the Hu/Monagan GCD algorithm from [7]. The timings in column MGCD2 are for our improved algorithm which interpolates the smaller of $Kr(H)$ and $Kr(C)$. The timings in column eval are are

the total time in MGCD2 spent evaluating polynomials in line 6 of PGCD and line 10 of SGCD.

		Maple timings		New GCD timings				
$\#G$	$\#\bar{A}, \#\bar{B}$	factor	gcd	MGCD1	t	MGCD2	t	eval
10^1	10^3	3.312s	1.77s	0.078s	3	0.113s	3	0.075s
10^2	10^2	5.161s	4.32s	0.190s	26	0.164s	19	0.086s
10^3	10^1	3.467s	40.83s	1.412s	213	0.114s	3	0.023s
10^1	10^4	11.81s	21.59s	0.661s	5	0.395s	3	0.059s
10^2	10^3	19.63s	47.74s	0.731s	18	0.707s	19	0.321s
10^3	10^2	23.33s	295.4s	3.852s	201	1.262s	22	0.447s
10^4	10^1	14.36s	11084.s	45.00s	2112	1.450s	2	0.058s
10^1	10^5	135.6s	331.1s	7.32s	5	5.35s	3	0.382s
10^2	10^4	148.6s	2413.s	10.95s	19	6.90s	24	2.636s
10^3	10^3	325.2s	31952.s	30.49s	198	25.56s	197	20.36s
10^4	10^2	138.2s	NA	238.2s	2063	13.15s	23	3.517s
10^5	10^1	97.1s	NA	3511.s	21037	10.47s	3	0.596s
10^1	10^6	12876.6s	5740.s	179.4s	5	180.6s	3	4.21s
10^2	10^5	17723.3s	NA	317.9s	17	94.5s	17	18.26s
10^3	10^4	9985.4s	NA	585.9s	186	260.2s	201	191.6s
10^4	10^3	8460.3s	NA	3128.7s	2132	397.2s	204	289.5s
10^5	10^2	8292.8s	NA	25612.s	21043	125.6s	22	34.15s
10^6	10^1	16537.7s	NA	NA	–	109.9s	2	4.22s

Table 1: Timings in CPU seconds for sparse GCD problems in 8 variables

The timing data shows that as $\#G$ increases the cost of Maple's GCD algorithm and our new algorithm (column MGCD1) increase.

For $\#G = \#\bar{A} = 10^3$ Maple took 31,952s and MGCD2 took 25.56s for a speedup of a factor of over 1000. For $\#G = 10^4$ and $\#\bar{A} = 10$ Maple took 11,084s and MGCD2 took 1.45s for a speedup of a factor of over 7000. The two columns labelled t are the number of terms in the largest polynomial in y interpolated by Algorithm ModifiedBT. There is some irregularity in the timings which is caused by Maple spending too much time (over 80% of the time) in garbage collection. For example, the time of 180.6s for MGCD2 in row 13 includes 148.2s of garbage collection time. This problem is being addressed by Maplesoft.

We included timings for Maple's factor command as a "sanity check". If the time for gcd is slower than factor then something is wrong as we could simply use the factorization of A and B to determine the gcd G. As the reader can see Maple's factor command is faster than our MGCD1 timings when the size of $\bar{A}$ and $\bar{B}$ is smaller than $\#G$. This observation is what forced us to redesign the Hu/Monagan gcd algorithm to interpolate the smaller of $G, \bar{A}, \bar{B}$. We are pleased to see that the timings for MGCD2 are always faster than the timings for factor.

6 Conclusion

We have developed and implemented a new polynomial GCD algorithm which improves on the Hu/Monagan GCD algorithm from [7, 8] by interpolating simultaneously a multiple H of G and a multiple C of $\bar{A}$. This reduces the cost dramatically when $\#\bar{A} \gg \#G$.

Our Maple implementation, with several sub-algorithms coded in C for increased speed, performs well; it is much faster than Maple's current GCD algorithm which is an implementation of Zippel's sparse modular GCD algorithm. Will it always be faster than Zippel's algorithm?

In order to invert the Kronecker substitution Kr we need the prime p_1 chosen by Algorithm PGCD to satisfy $p_1 > \prod_{i=1}^{n} r_i$. If $n = 20$ and $d_i = \deg(A, x_i) = 20$ then $r_i > d_i$ forces $p_1 > 21^{20} = 2^{87.8}$ which is too big for us to use the 64 bit integer arithmetic available on our computers to do the modular arithmetic in $\mathbb{Z}_p$. Our GCD algorithm will use a 128 bit prime so all arithmetic will be done in software leading to a large slow down. In [6] Lecerf and van der Hoeven investigate methods for using smaller r_i. We will explore this.

References

[1] N. B. Atti, G. M. Diaz-Toca, and H. Lombardi. The Berlekamp-Massey algorithm revisited. AAECC **17**:75–82, 2006.

[2] Michael Ben-Or and Prasoon Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proc. of STOC '20*, pp. 301–309, ACM, 1988.

[3] W. S. Brown. On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors. *J. ACM* **18**:478–504, 1971.

[4] G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Math. Comp.* **36**(154):587–592, 1981.

[5] K.O. Geddes, G. Labahn, S. Czapor. *Algorithms for Computer Algebra* Kluwer Academic, 1992.

[6] Joris van der Hoeven and Grégoire Lecerf. Sparse Polynomial Interpolation in Practice. *Communications in Computer Algebra* **48**(4):187–191, 2015.

[7] Jiaxiong Hu and Michael Monagan. A fast parallel sparse polynomial GCD algorithm. In *Proc. of ISSAC 2016*, pp. 271–278, ACM, 2016.

[8] Jiaxiong Hu and Michael Monagan. A Fast Parallel Sparse Polynomial GCD Algorithm. *J. Symb. Cmpt.* **105**:(1) 28–63, Springer, July 2021.

[9] J. de Kleine, M. Monagan, A. Wittkopf. Algorithms for the non-monic case of the Sparse Modular GCD Algorithm. *Proc. of ISSAC '2005*, pp. 124–131, ACM, 2005.

[10] J. L. Massey. Shift-register synthesis and BCH decoding. *IEEE Trans. on Information Theory*, 15:122–127, 1969.

[11] E. Kaltofen, W. Lee, and A. Lobo. Early Termination in Ben-Or/Tiwari Sparse Interpolation and a Hybrid of Zippel's algorithm. In *Proc. ISSAC 2000*, pp. 192–201, ACM, 2000.

[12] E. Kaltofen. Fifteen years after DSC and WLSS2. In *Proc. of PASCO 2010*, pp. 10–17, ACM, 2010.

[13] Niels Möller and Torbjorn Grandlund. Improved Division by Invariant Integers. *Transactions on Computers* **60**(2):165–175, IEEE, 2011.

[14] Michael Monagan and Roman Pearce. POLY: A new polynomial data structure for Maple 17. In *Computer Mathematics*, pp. 325–348, Springer, 2014. Ruyong Feng, Wen-shin Lee, Yosuke Sato editors.

[15] Michael Monagan and Roman Pearce. The design of Maple's sum-of-products and POLY data structures for representing mathematical objects. *Communications in Computer Algebra*, **48**(4):166–186, ACM, 2014.

[16] Michael Monagan and Baris Tuncer. Using Sparse Interpolation in Hensel Lifting. *Proc. of CASC 2016*, LNCS **9890**:381–400, Springer, 2016.

[17] Michael Monagan and Baris Tuncer. Factoring multivariate polynomials with many factors and huge coefficients. *Proc. of CASC 2018*, LNCS **11077**:319–334, Springer, 2018.

[18] Michael Monagan and Baris Tuncer. Polynomial Factorization in Maple 2019. In *Proc. of the 2019 Maple Conference. Maple in Mathematics Education and Research.* CCIS **1125**:341–345, Springer, 2020.

[19] Joel Moses and David Y.Y. Yun. The EZ GCD algorithm. *Proc. ACM '73*, 159–166. ACM, 1973.

[20] Hirokazu Murao and Tetsuro Fujise. Modular Algorithm for Sparse Multivariate Polynomial Interpolation and its Parallel Implementation. *J. Symb. Cmpt.* **21**:377–396, 1996.

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

[22] Y. Sugiyama, M. Kashara, S. Hirashawa and T. Namekawa. A Method for Solving Key Equation for Decoding Goppa Codes. *Information and Control* **27**:87–99, 1975.

[23] Paul S. Wang. The EEZ-GCD algorithm. *ACM SIGSAM Bulletin* **14**(2):50–60, 1980.

[24] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *Proc. of EUROSAM '79*, pp. 216–226, Springer, 1979.

[25] Richard Zippel. Interpolating Polynomials from their Values. *J. Symb Cmpt.* **9**:375–403, Springer, 1990.