Article Submitted to Journal of Symbolic Computation

On the Modular Polynomial GCD Algorithm over the Integers, Finite Fields, and Number Fields

Erich Kaltofen[1], Michael Monagan[2] and Allan Wittkopf[2]

[1] *Mathematics Department, North Carolina State University, Box 8205, Raleigh, N.C. 27695-8205 USA.*
E-mail: `kaltofen@math.ncsu.edu`

[2] *Centre for Experimental and Constructive Mathematics, Simon Fraser University, Burnaby, British Columbia, V5A 1S6 Canada.*
E-mail: `monagan@cecm.sfu.ca`, `wittkopf@cecm.sfu.ca`

Abstract

We study the design and implementation of the modular polynomial GCD algorithm of Brown. We present an improved design for $\mathbb{Z}[x, y]$ and compare it asymptotically with Brown's original algorithm, the heuristic GCD algorithm GCDHEU, and with the EEZ-GCD algorithm. We also make an empirical comparison based on Maple implementations of the algorithms. Our findings show that a careful implementation of our improved version of Brown's algorithm is much better than the other algorithms in theory and in practice. We present also an improved design for small finite fields and for algebraic number fields of low degree.

1. Introduction

Efficient computation of greatest common divisors (GCDs) of multivariate polynomials is very important in computer algebra systems because the GCD operation is the bottleneck of many basic applications. Whereas the classical algorithms for multivariate polynomial multiplication and division are effective for many applications, this is not the case with computing GCDs. Thirty years ago Brown (4) developted the modular GCD algorithm, which solved the problem of efficient GCD computation in $\mathbb{Z}[x_1, ..., x_n]$ when the input polynomials are

*This material is based on work supported in part by the National Science Foundation under Grant No. CCR-9712267 (Kaltofen), by NSERC of Canada (Monagan) and by the MITACS National Centre of Excellence (Monagan).

dense. This was a major breakthrough for it eliminated the problem of intermediate expression swell that occurs in the family of algorithms based on the Euclidean algorithm which, in the multivariate case, often result in a GCD computation "blowing up". However, Brown's algorithm is not effective for sparse polynomials in many variables. This problem was solved by Zippel (17**?**) in 1979. Many other methods have also been developed for efficient GCD computation in $\mathbb{Z}[x_1, \dots, x_n]$. Hensel lifting can be utilized in what is called the EZ-GCD algorithm (see (**?** , section 7) and the papers by Moses, Wang, and Yun cited there) and a variation on it by called the EEZ-GCD algorithm (see (16)). It is possible to compute the GCD of two polynomials viewed a black boxes for their evaluation again as a black box (**? ?**). Another algorithm which reduces a polynomial GCD problem to a single large integer GCD calculation is the heuristic GCD algorithm of Char et. al. (5) which is effective for dense polynomials. The EZ-GCD and EEZ-GCD algorithms appear to be the most widely used algorithms. It is the default for multivariate polynomials over $\mathbb{Z}$ in the REDUCE, Macsyma, Axiom and MAGMA computer algebra systems. It is also used by Maple on problems with 3 or more variables when the GCDHEU algorithm fails. Zippel's algorithm is used by Mathematica and is available as an option in Macsyma. Apparently Brown's algorithm is not widely used. It is available in Macsyma as an option, it is used by MAGMA for GCDs over number fields, and is used in Maple on problems with 2 variables when the GCDHEU algorithm fails. Nevertheless, it is the most effective algorithm for polynomials which are not sparse. We suppose three reasons for why it is not in common usage. Firstly, it is more difficult to implement properly. Secondly, there was a bias on the part of the system designers in the 1970s and 1980s towards sparse algorithms which overlooked the fact that large GCD problems in a few variables are often not sparse. Thirdly, Brown's original design is poor on problems which have a small GCD.

The modular algorithms can be readily parallelized (**?**).

The purpose of this paper is to take another look at the modular algorithm of Brown in the context of bivariate GCD computations over the integers, finite fields and algebraic number fields. Our interest in bivariate GCDs resulted from GCD problems given to us by Edgardo Cheb-Terrab (in the context of finding invariants of Abel ODE, see (6)), and David Boyd (in the context of computing A-polynomials). These GCD problems are characterized by being large and not sparse. Also in the context of factoring bivariate polynomials over small finite fields Laurent Bernardin (1) has given us another source of large bivariate GCD problems, mostly over GF(2).

In section 2 we review Brown's original algorithm and describe a simpler and a more common design of it in which termination is done using trial division. In section 3 we detail an improved version of Brown's algorithm which is always efficient when the GCD is small. We give an asymptotic analysis for it, Brown's algorithm, the EEZ-GCD algorithm and the heuristic algorithm GCDHEU, and compare Maple implementations of our improved version with Maple's implementation of the EEZ-GCD algorithm and GCDHUE. Our contention is that most bivariate GCD problems in practice are not sparse enough to warrant the use of a sparse GCD algorithm like the EEZ-GCD algorithm and since our modified version of Brown's algorithm is so much faster than GCDHEU it is the algorithm of choice. In section 4 we use our improved version of Brown's algorithm to compute bivariate GCDs over number fields. We find that in order to obtain good performance when a number field has low degree, we must map the computation into $\mathbb{Z}_p$. Secondly we describe the use of the modular GCD algorithm for GCDs over small finite fields and give an asymptotic analysis for it.

1.1. Notation

Throughout the paper we will use the following notations:

$||a||_\infty$ denotes the height of a polynomial, that is, the magnitude of the largest integer coefficient.

$\mathrm{cont}_X : U[X] \to U$ is the content of a polynomial, namely the GCD of the coefficients in U, a unique factorization domain.

$\mathrm{pp}_X : U[X] \to U[X]$ is the primitive-part of a polynomial, namely the polynomial divided by it's content.

We define the following notations for multivariate operations that depend on a term ordering on the variables $X = x_1, x_2, \ldots, x_n$. In our implementations we always use lexicographical ordering but in principal, any admissible term ordering may be used.

$\mathrm{lc}_X : R[X] \to R$ is the leading coefficient of a polynomial using lexicographical ordering on the variables X.

$\deg_X : R[X] \to \mathbb{N}^n$ is the vector degree of a polynomial using lexicographical order.

$\min_X : (\mathbb{N}^n, \mathbb{N}^n) \to \mathbb{N}^n$ is the minimum of the two terms using lexicographical order.

2. The Modular GCD Algorithm for $\mathbb{Z}[x_1, x_2, \ldots, x_n]$

2.1. Brown's GCD Algorithm (1971)

Brown's description of the "modular GCD algorithm" in (4) is reproduced here in sufficient detail for the purpose of understanding and asymptotic analysis. Brown's algorithm has two subroutines, M and P. The main subroutine M maps a polynomial GCD computation in $\mathbb{Z}[x_1, x_2, \ldots, x_n]$ into one or more GCD computations in $\mathbb{Z}_p[x_1, x_2, \ldots, x_n]$ for primes $p \in p_0, p_1, \ldots \in \mathbb{Z}$ and applies the Chinese remainder theorem to the resulting GCD images. Brown's presentation of subroutine P is similar. It maps a polynomial GCD computation in $\mathbb{Z}_p[x_1, x_2, \ldots, x_n]$ into one or more GCD computations in $\mathbb{Z}_p[x_1, x_2, \ldots, x_{n-1}]$ by evaluating the input polynomials at $x_n = \alpha$ for sufficiently many $\alpha \in \mathbb{Z}_p$ and recovers x_n, again, by application of the Chinese remainder theorem.

Algorithm M (Brown 1971)

Input: $a, b \in \mathbb{Z}[X = x_1, x_2, \ldots, x_n] \setminus \{0\}$
Output: $g = \mathrm{GCD}(a, b)$, $\bar{a} = a/g$, $\bar{b} = b/g$

1 Compute integer contents and their GCD.
$ca, cb = \mathrm{cont}_X(a), \mathrm{cont}_X(b)$, and $cg = \mathrm{GCD}(ca, cb)$.
Set $(a, b) = (a/ca, b/cb)$.

2 Compute $\gamma \in \mathbb{Z}$ the correction coefficient.
Set $la = \mathrm{lc}_X(a)$, $lb = \mathrm{lc}_X(b)$, and $\gamma = \mathrm{GCD}(la, lb)$.

3 Estimate twice the height of $\max(\gamma \cdot g, \gamma \cdot \bar{a}, \gamma \cdot \bar{b})$.
Set $B = 2\,\gamma\, \max(||a||_\infty, ||b||_\infty)$.

4 Set $m = 1$ the product of the moduli.

5 Set $\mathbf{e} = \min(\deg_X a, \deg_X b)$ bounding $\deg_X g$.

Loop 6 Choose a new prime p that does not divide $la \times lb$.

7 Compute $(a_p, b_p, \gamma_p) = (a, b, \gamma) \bmod p$.

8 Compute $g_p, \bar{a}_p, \bar{b}_p \in \mathbb{Z}_p[X]$ the monic GCD of $\gamma_p \cdot a_p$ and $\gamma_p \cdot b_p$ and their cofactors using algorithm P. If P fails (too few evaluation points are available in $\mathbb{Z}_p$) then algorithm M fails also.

9 Test for a, b relatively prime.
If $\deg_X g_p = \mathbf{0}$ then output cg, $(ca/cg)a$, $(cb/cg)b$.

10 If $\deg_X g_p > \mathbf{e}$ then this prime is unlucky so goto Loop.

11 Leading coefficient correction in $\mathbb{Z}$.
Set $(g_p,\ \bar{a}_p,\ \bar{b}_p) = (\gamma \cdot g_p,\ \gamma^{-1} \cdot a_p,\ \gamma^{-1} \cdot b_p) \bmod p$.

12 If $m = 1$ then (first image) set $(g_m,\ \bar{a}_m,\ \bar{b}_m) = (g_p,\ \bar{a}_p,\ \bar{b}_p)$, $m = p$, $\mathbf{e} = \deg_X g_p$ and goto Loop.

13 If $\deg_X g_p < \mathbf{e}$ then all previous primes were unlucky so restart the algorithm keeping only the current prime.
Set $(g_m,\ \bar{a}_m,\ \bar{b}_m) = (g_p,\ \bar{a}_p,\ \bar{b}_p)$, $m = p$, $\mathbf{e} = \deg_X g_p$ and goto Loop.

14 Combine the new images with the old using Chinese remaindering and express the result in the symmetric range for $\mathbb{Z}_p$. Set
$g_m = \mathrm{CRA}([g_p, g_m], [p, m])$,
$\bar{a}_m = \mathrm{CRA}([\bar{a}_p, \bar{a}_m], [p, m])$, and
$\bar{b}_m = \mathrm{CRA}([\bar{b}_p, \bar{b}_m], [p, m])$.

15 Set $m = m \times p$. If $m \leq B$ then goto Loop.

16 Test for termination. Without explicitly computing $g_m \times \bar{a}_m$ and $g_m \times \bar{b}_m$ determine if $||g_m \times \bar{a}_m||_\infty < m/2$ and $||g_m \times \bar{b}_m||_\infty < m/2$. If so then we

have $m|(a - g_m \times \bar{a}_m)$ and $m|(b - g_m \times \bar{b}_m)$ and also $\gamma \cdot a - g_m \times \bar{a}_m = 0$ and $\gamma \cdot b - g_m \times \bar{b}_m = 0$ over $\mathbb{Z}$ hence we are done; so set
$g = \mathrm{pp}_X(g_m)$, $lg = \mathrm{lc}_X g \in \mathbb{Z}$, $\bar{a} = \bar{a}_m/lg$, $\bar{b} = \bar{b}_m/lg$ and
output $cg \cdot g$, $(ca/cg) \cdot \bar{a}$, $(cb/cg) \cdot \bar{b}$.

17 Goto Loop – either we do not have enough primes yet or all primes are unlucky.

An important observation to make about Brown's design of the modular GCD algorithm is that both subroutines have been deliberately designed to terminate when $\gamma \cdot a - g_m \times \bar{a}_m = 0$ and $\gamma \cdot b - g_m \times \bar{b} = 0$ without explicit multiplication of $g_m \times \bar{a}_m$ and $g_m \times \bar{b}_m$. The integer $\gamma = \mathrm{GCD}(\mathrm{lc}_X(a), \mathrm{lc}_X(b)) = \alpha \cdot \mathrm{lc}_X(g)$. The input polynomials a and b, after being made primitive in step 1 are implicitly multipied by γ prior to invoking algorithm P to compute the GCD in step 8. At termination in step 16 we will have $g_m = \alpha \cdot g, \bar{a}_m = \mathrm{lc}_X(g) \cdot \bar{a}$ and $\bar{b}_m = \mathrm{lc}_X(g) \cdot \bar{b}$ with $\mathrm{lc}_X(g_m) = \mathrm{lc}_X(\bar{a}_m) = \mathrm{lc}_X(\bar{b}_m) = \gamma$. This design leads to good performance on some GCD problems but poor performance on others.

Algorithm P (below) computes a GCD in $\mathbb{Z}_p[x_1, \dots, x_n]$ by evaluating the input polynomials at $x_n = \alpha$ chosen at random from $\mathbb{Z}_p$, computes this GCD in one less variable using algorithm P recursively, does this for different α then interpolates x_n from the images. For it to terminate successfully, p will have to be large enough so that sufficiently many evaluation points exist to interpolate x_n in $\alpha \cdot g, \mathrm{lc}_X(g) \cdot \bar{a}, \mathrm{lc}_X(g) \cdot \bar{b}$. Appart from this, there are no further constraints on the choice of primes in step 6 of algorithm M. However, the choice of primes is critical for the overall efficiency of an implementation of Brown's algorithm. In principle, one could use one very large prime $p > 2B$. However, one order of magnitude in efficiency can be gained by using many small primes instead of one large prime, that is to say, application of the Chinese remainder theorem essentially buys us one order of magnitude. Additionally, if the primes are chosen so that arithmetic in $\mathbb{Z}_p$ can be done directly in the hardware of the machine then there will be a very sizable, though not asymptotic, improvement in efficiency. The best choice then is to choose the largest primes that fit in the word length of the machine, which is typically 32 or 64 bits.

Brown views evaluation at $x_n = \alpha$ as equivalent to computing modulo $x_n - \alpha$ so that after sufficiently many image GCDs have been computed in $\mathbb{Z}_p[x_1, \dots, x_{n-1}]$ the coefficients in x_n are again recovered by Chinese remaindering. Replacing the Euclidean coefficient domain $\mathbb{Z}$ in Algorithm M by the Euclidean domain $\mathbb{Z}_p[x_n]$, that is, with inputs $a, b \in \mathbb{Z}_p[x_n][x_1, \dots, x_{n-1}]$, this second stage of the modular GCD algorithm, algorithm P, becomes almost equivalent to the first stage where the primes p chosen in step 6 of algorithm M are now the linear polynomials $x_n - \alpha$. In reading the steps of algorithm P the reader should refer back to the corresponding step in algorithm M.

Algorithm P (Brown, 1971)

Input: $a, b \in \mathbb{Z}_p[x_n][X = x_1, \ldots, x_{n-1}] \setminus \{0\}$, p prime.
Output: $g = \mathrm{GCD}(a, b)$, $\bar{a} = a/g$, $\bar{b} = b/g$

0 If $n = 1$ then compute g=GCD(a, b) using the Euclidean algorithm and output $(g, a/g, b/g)$.

1 Compute contents in $\mathbb{Z}_p[x_n]$ and their GCD.
Set $ca, cb = \mathrm{cont}_X(a), \mathrm{cont}_X(b)$, $cg = \mathrm{GCD}(ca, cb)$, and $(a, b) = (a/ca, b/cb)$.

2 Compute $\gamma \in \mathbb{Z}_p[x_n]$ the correction coefficient.
Set $la = \mathrm{lc}_X(a)$, $lb = \mathrm{lc}_X(b)$, and $\gamma = \mathrm{GCD}(la, lb)$.

3 Bound the degree in x_n of $\gamma \times g$, $\gamma \times \bar{a}$, $\gamma \times \bar{b}$.
Set $B = \deg_{x_n} \gamma + \max(\deg_{x_n} a, \deg_{x_n} b)$.

4 Set $m = 1$ the product of the moduli.

5 Set $\mathbf{e} = \min(\deg_X(a), \deg_X b)$.

Loop 6 Choose $\alpha \in \mathbb{Z}_p$ such that $x_n - \alpha$ does not divide $la \times lb$. If no such evaluation point exists then p is too small and the algorithm fails.

7 Compute $(a_\alpha, b_\alpha, \gamma_\alpha) = (a, b, \gamma) \bmod x_n - \alpha$.

8 Invoke algorithm P recursively to compute $g_\alpha, \bar{a}_\alpha, \bar{b}_\alpha \in \mathbb{Z}_p[X]$ the monic GCD of $\gamma_\alpha \cdot a_\alpha$ and $\gamma_\alpha \cdot b_\alpha$ and their cofactors.

9 Test for a, b relatively prime.
If $\deg_X g_\alpha = \mathbf{0}$ then output cg,$(ca/cg)a$,$(cb/cg)b$.

10 If $\deg_X g_\alpha > \mathbf{e}$ then (skip this prime) goto Loop.

11 Leading coefficient correction in $\mathbb{Z}_p[x_n]$.
Set $(g_\alpha, \bar{a}_\alpha, \bar{b}_\alpha) = (\gamma(\alpha) \cdot g_\alpha, \gamma(\alpha)^{-1} \cdot \bar{a}_\alpha, \gamma(\alpha)^{-1} \cdot \bar{b}_\alpha) \bmod p$.

12 If $m = 1$ then (first image) then
Set $(g_m,\ \bar{a}_m,\ \bar{b}_m) = (g_\alpha,\ \bar{a}_\alpha,\ \bar{b}_\alpha)$, $m = x_n - \alpha$, $\mathbf{e} = \deg_X g_\alpha$ and goto Loop.

13 If $\deg_X g_\alpha < \mathbf{e}$ then all previous points were unlucky so restart the algorithm keeping only the current point.
Set $(g_m,\ \bar{a}_m,\ \bar{b}_m) = (g_\alpha,\ \bar{a}_\alpha,\ \bar{b}_\alpha)$, $m = x_n - \alpha$, $\mathbf{e} = \deg_X g_\alpha$ and goto Loop.

14 Combine the new image with the old using the CRA. Set
$g_m = \mathrm{CRA}([g_\alpha, g_m], [x_n - \alpha, m])$, $\bar{a}_m = \mathrm{CRA}([\bar{a}_\alpha, \bar{a}_m], [x_n - \alpha, m])$, and
$\bar{b}_m = \mathrm{CRA}([\bar{b}_\alpha, \bar{b}_m], [x_n - \alpha, m])$.

15 Set $m = m \times (x_n - \alpha)$. If $\deg_{x_n} m \leq B$ then goto Loop.

16 Test for termination.
If $\deg_{x_n} g_m + \deg_{x_n} \bar{a}_m = \deg_{x_n} \gamma + \deg_{x_n} a$ and $\deg_{x_n} g_m + \deg_{x_n} \bar{b}_m = \deg_{x_n} \gamma + \deg_{x_n} b$ then set $g = \mathrm{pp}_X(g_m)$, $lg = \mathrm{lc}_X g$, $\bar{a} = \bar{a}_m/lg$, $\bar{b} = \bar{b}_m/lg$, and output $cg \times g$, $(ca/cg)\bar{a}$, $(cb/cg)\bar{b}$.

17 Goto Loop – either all α are unlucky or we do not have enough yet.

The complexity of Brown's algorithm will depend on the number of primes used by subroutine M and the number of evaluation points used by algorithm P. The reader will note that these quantities are determined by the bounds computed in step [3] of both algorithms, and that they are a function of the size of the inputs a and b and not the size of $g, \bar{a}$ and $\bar{b}$. In order to understand then improve the asymptotic time complexity of Brown's algorithm we first identify two classes of GCD problems which are common in practice. For this analysis we restrict our attention to bivariate GCD problems in $\mathbb{Z}[x, y]$.

Balanced GCD Problems

Suppose $||\bar{a}||_\infty, ||\bar{b}||_\infty$ and $||g||_\infty < 10^n$. Suppose also that $\deg_x \bar{a} = \deg_x \bar{b} = \deg_x g = n$ and $\deg_y \bar{a} = \deg_y \bar{b} = \deg_y g = n$. Thus in this class of GCD problems we assume that the size of the coefficients grows proportionately with the degree of the polynomials in both variables. If the polynomials $g, \bar{a}$ and $\bar{b}$ are also dense then their size is $\mathrm{O}(n^3)$ and the cost of computing $\bar{a} \times g$ or a/g, using classical algorithms for integer and polynomial multiplication and division, is $\mathrm{O}(n^6)$. Suppose instead that the polynomials $g, \bar{a}$ and $\bar{b}$ have n terms instead of $\mathrm{O}(n^2)$. We will call such polynomials semi-sparse. Their size will be $\mathrm{O}(n^2)$ and the cost of the classical multiplication and division algorithms reduce to $\mathrm{O}(n^4)$. The reason to consider semi-dense polynomials is because these two classes, dense and semi-dense, bracket many practical problems in the sense that the number of non-zero terms is typically greater than n.

Small GCD Problems

Suppose $||\bar{a}||_\infty < 10^n$ and $||\bar{b}||_\infty < 10^n$ and $\deg_x \bar{a} = \deg_y \bar{a} = n$ and $\deg_x \bar{b} = \deg_y \bar{b} = n$. Suppose $||g||_\infty = O(1)$ and $\deg_x g = \deg_y g = O(1)$. Thus for this problem we assume that for the cofactors, the size of the coefficients grows proportionately with the degree in both variables but the size of the GCD remains fixed. We are thinking of the case when the size of the GCD is small e.g. degree 2 in x and y but the size of the cofactors is large, e.g. degree 18 in x and y. If $\bar{a}$ and $\bar{b}$ are dense then the cost of computing $\bar{a} \times g$ using the classical algorithms for polynomial multiplication and long division is $\mathrm{O}(n^3)$. If the polynomials $\bar{a}$ and $\bar{b}$ are semi-sparse, each having $\mathrm{O}(n)$ terms, then their size is $\mathrm{O}(n^2)$. The cost of multiplication and division is $\mathrm{O}(n^2)$. We will show in section 3 that the asymptotic complexity of Brown's algorithm is $\mathrm{O}(n^4)$ for the balanced and small GCD problems. This complexity is good for the balanced GCD case but poor for the small GCD case.

2.2. The Trial Division Modular GCD Algorithm

Instead of lifting all of $g, \bar{a}, \bar{b}$ and stopping when $a - g\bar{a} = 0$ and $b - g\bar{b} = 0$, one may instead lift g one image at a time stopping if $g|a$ and $g|b$. By performing these *trial divisions* only when g does not change from one image to the next, one will, with high probability, make the trial divisions once. This means that algorithm M almost always uses one more prime than necessary to reconstruct $\alpha \cdot g$ and obtains $\bar{a}$ and $\bar{b}$ by division. The same problem occurs in subroutine P and one can employ the same idea to resolve it. NEW REMARKS This idea is well known. It is mentioned in the text in the chapter "Computing by Homomorphic Images" by Lauer in (3) in 1982. George Labahn's presentation of Brown's algorithm in section 7.4 of Geddes et. al. (**?**) uses trial division for termination. Indeed Richard Fateman informed us that his implementation of Brown's algorithm in Macsyma, which he made just prior to Brown's paper appearing, uses trial division.

We present here the modified versions of algorithm M and P which lift only the GCD g and terminate using trial division. For notational convenience we define $\phi : \mathbb{Z} \to \mathbb{Z}_p$ to be the modular ring morphism $\phi(a) = a \bmod p$. When applied to a polynomial we mean it is applied to the coefficients of the polynomial.

Algorithm M— with Trial Division

Input: $a, b \in \mathbb{Z}[X = x_1, x_2, \ldots, x_n] \setminus \{0\}$
Output: $g = \mathrm{GCD}(a, b)$, $\bar{a} = a/g$, $\bar{b} = b/g$

0 If $n = 0$ then $g \leftarrow \mathrm{GCD}_{\mathbb{Z}}(a, b)$ and output $g, a/g, b/g$.

1 $(ca, cb) \leftarrow (\mathrm{cont}_X(a), \mathrm{cont}_X(b))$, $c \leftarrow \mathrm{GCD}_{\mathbb{Z}}(ca, cb)$.

2 $(a, b) \leftarrow (a/ca, b/cb)$,

3 $\gamma \leftarrow \mathrm{GCD}_{\mathbb{Z}}(\mathrm{lc}_X(a), \mathrm{lc}_X(b))$.

4 $\mathbf{e} \leftarrow \min(\deg_X a, \deg_X b)$.

5 $(m, g_m) \leftarrow (1, 0)$.

Loop 6 Choose a prime $p \in \mathbb{Z}$ such that $p \nmid m\gamma$.

7 $g_p \leftarrow \mathrm{GCD}(\phi_p(a), \phi_p(b))$ using algorithm P—.

8 If $\deg(g_p) =_X \mathbf{0}$ then output c and stop.

9 If $\deg(g_p) >_X \mathbf{e}$ then (skip this prime) goto Loop.

10 $g_p \leftarrow \gamma \cdot \mathrm{lc}_X(g_p)^{-1} \cdot g_p$ in $\mathbb{Z}/\langle p\rangle[X]$.

11 If $\deg(g_p) <_X \mathbf{e}$ then $(m, g_m, \mathbf{e}) \leftarrow (1, 0, \deg(g_p))$.

12 $h \leftarrow \mathrm{CRA}^*([g_p, g_m], [p, m])$.

13 If $h = g_m$ then begin termination test.

14 $\quad g \leftarrow h/\mathrm{cont}_X(h)$.

*Note, the result computed by the Chinese remainder algorithm (CRA) in step 12 must be expressed in the symmetric range of the integers modulo $p \cdot m$ so that negative integer coefficients in g can be reconstructed as well as positive ones. We will return to this issue when we try to generalize the algorithm in section 4.

15 If $g \mid a$ and $g \mid b$ then output $c \cdot g, (ca/c) \cdot (a/g), (cb/c) \cdot (b/g)$

16 $(m, g_m) \leftarrow (p \cdot m, h)$.

17 Goto Loop.

The main elements of the proof of correctness of the algorithm are reproduced here for discussion later when we look at the general setting of the algorithm. The requirement that p not divide γ in step (6) means that Lemma 1 below holds. Lemma 1 tells us the relationship between the image GCD g_p that we compute in step (7) and what we want to compute, namely $\phi_p(g)$. For later reference, we prove this Lemma for coefficients in any Euclidean ring E in place of $\mathbb{Z}$. The function $\mathrm{lc}_X : E[X] \to E$ denotes the leading coefficient operation of a polynomial under an admissible term ordering on $X = x_1, \dots, x_n$. It satisfies $\mathrm{lc}_X(a \times b) = \mathrm{lc}_X(a) \times \mathrm{lc}_X(b)$.

Lemma 1: *Let E be a Euclidean ring, $p \in E$ be a prime and $\phi_p : E \to E/\langle p\rangle$ be the modular ring morphism $\phi(a) = a \bmod p$. The natural extension of ϕ_p to $E[x_1, \dots, x_n]$ where ϕ_p is applied to the coefficients of the polynomial is also a ring morphism.*

Let $a, b \in E[x_1, \dots, x_n] \setminus \{0\}$. Let g be any greatest common divisor of a and b. Thus $a = g\bar{a}$ and $b = g\bar{b}$ for some $\bar{a}, \bar{b} \in E[x_1, \dots, x_n]$. Let $g_p = \mathrm{GCD}(\phi_p(a), \phi_p(b))$. If $\phi_p \mathrm{lc}_X(g) \neq 0$ then

(i) $g_p = \delta \cdot \phi_p(g) \times \Delta$ for some $\delta \in E/\langle p\rangle$ and $\Delta = \mathrm{GCD}(\phi_p(\bar{a}), \phi_p(\bar{b}))$, and

(ii) $\deg_X(g_p) \geq_X \deg_X(g)$.

Proof. Note that the two rings $E[x_1, \dots, x_n]$ and $E/\langle p\rangle[x_1, \dots, x_n]$ are both unique factorization domains as the residue ring $E/\langle p\rangle$ is a field. If follows that GCDs in both rings are unique up multiplication by units in E and $E/\langle p\rangle$ respectively.

Since ϕ is a ring morphism it follows that $\mathrm{GCD}(\phi_p(a), \phi_p(b)) = \mathrm{GCD}(\phi_p(g\bar{a}), \phi_p(g\bar{b})) = \mathrm{GCD}(\phi_p(g) \cdot \phi_p(\bar{a}), \phi_p(g) \cdot \phi_p(\bar{b}))$. Since $E/\langle p\rangle[x_1, \dots, x_n]$ is a unique factorization domain then

$$\mathrm{GCD}(\phi_p(g) \cdot \phi_p(\bar{a}), \phi_p(g) \cdot \phi_p(\bar{b})) = \phi_p(g) \cdot lc_X(\phi_p(c))^{-1} \times \mathrm{GCD}(\phi_p(\bar{a}), \phi_p(\bar{b})).$$

Hence $g_p = \delta \cdot \phi_p(g) \times \Delta$ as claimed in (i). Now $\phi_p(\mathrm{lc}_X(g)) \neq 0 \implies \deg_X(\phi_p(g)) =_X \deg_X(g)$. Since $\mathrm{GCD}(\bar{a}, \bar{b}) = 1$ then $\phi_p(\bar{a})$ and $\phi_p(\bar{b})$ cannot be both zero hence $\mathrm{GCD}(\phi_p(\bar{a}), \phi_p(\bar{b})) \neq 0$, thus (ii) follows. ⊠

Because $\gamma = GCD_{\mathbb{Z}}(\mathrm{lc}_X(a), \mathrm{lc}_X(b))$ and $\mathrm{lc}_X(a) = \mathrm{lc}_X(g) \cdot \mathrm{lc}_X(\bar{a})$ and $\mathrm{lc}_X(b) = \mathrm{lc}_X(g) \cdot \mathrm{lc}_X(g))$ it follows that $\phi_p(\gamma) \neq 0 \implies \phi_p(\mathrm{lc}_X(g)) \neq 0$ hence Lemma 1 holds for the GCD computation in step 7 in algorithm M—. Therefore $\deg_X(g_p) \geq_X \deg_X(g)$, hence, if $\deg(g_p) =_X \mathbf{0}$ then $\deg(g) =_X \mathbf{0}$ which justifies the early termination of the algorithm in step 10.

If $\deg(\Delta) >_X \mathbf{0}$ then we say p is an *unlucky prime*. Clearly we will not be able to reconstruct g using g_p by Chinese remaindering if p is an unlucky prime, therefore, algorithm M must detect which primes are unlucky primes. This is done by applying Lemma 1 in steps 6, 9, and 11. Step 6 ensures that $\phi_p(\gamma) \neq 0$ so Lemma 1 holds. It follows from (ii) that either $\deg_X g_p =_X \deg_X g$, that is, p is not unlucky, or $\deg_X g_p >_X \deg_X(g)$, that is, p is unlucky. We say p is a *good prime* if p is not unlucky and $\phi_p(\gamma) \neq 0$. Now suppose in algorithm M that p is a *good prime*. Then if any previous primes were unlucky we have $\deg_X(g_m) >_X \deg_X(g_p)$ and these primes are indentified and removed from g_m in step 11. If any future prime q is unlucky, then we have $\deg_X(g_q) > \deg_X(g_p)$ and such a prime is identified and skipped in step 9. Note, in step 11 it could be that $\deg_X(g_m) > \deg_X(g_p) > \deg_X(g)$ meaning all previous primes are identified as unlucky but we do not know yet that the current prime p is also unlucky.

Now we argue that number of primes which are not good primes, is finite. Firstly, γ is an integer of finite size so only a finite number of primes are ruled out in step 6. Secondly, for the univariate case $\mathbb{Z}[x]$, the unlucky primes are characterized by those which divide $r = \mathrm{res}_x(\bar{a}, \bar{b})$. Since r is an integer of finite size, the number of unlucky primes is finite. Lemma 2 below establishes that the number of unlucky primes is finite for the multivariate case. We give this proof, which is based on a single resultant, as it allows the sharpest estimate on the number and size of unlucky primes, especially when combined with the inequalities in (**?**). However, we shall not analyze unluckiness further.

LEMMA 2: *The number of unlucky primes is finite.*

Proof. For later reference, we prove this Lemma for coefficients in a unique factorization domain U in place of $\mathbb{Z}$, and we shall count as one two prime elements p_1 and p_2 that are associates, i.e., if their quotient p_1/p_2 is an invertible element (a unit) in U. Note that in a UFD such as $\mathbb{Q}[x]$, where $\mathbb{Q}$ are the rational numbers, the number of units is infinite.

We first note a lemma by Gauss (see (8, Article 42), (**?**)) that relates a factorization in $\mathrm{QF}(U)[x_1, \ldots, x_n]$, where $\mathrm{QF}(U)$ is the field of quotients of U, to one in $U[x_1, \ldots, x_n]$. Suppose $A = a_1 \cdots a_r$, where $A \in U[x_1, \ldots, x_n]$ and $a_1, \ldots, a_r \in \mathrm{QF}(U)[x_1, \ldots, x_n]$. Then there exist $u_1, \ldots, u_r \in \mathrm{QF}(U)$ such that $A = (u_1 a_1) \cdots (u_r a_r)$ and $(u_1 a_1), \ldots, (u_r a_r) \in U[x_1, \ldots, x_n]$. Gauss's lemma actually justifies the above algorithm.

A prime p is unlucky exactly if $\mathrm{GCD}(\phi_p(\bar{a}), \phi_p(\bar{b})) \neq 1$, where $\bar{a}$ and $\bar{b}$ are the co-factors of a and b defined above. Since $\mathrm{GCD}(\bar{a}, \bar{b}) = 1$ in $U[x_1, \ldots, x_n]$, we have, by applying Gauss's lemma, $\mathrm{GCD}_{\mathrm{QF}(U)[x_1,\ldots,x_n]}(\bar{a}, \bar{b}) = 1$. Over the field $\mathrm{QF}(U)$ the GCD problem is a rational problem, which means that the GCD can be computed by arithmetic operations in $\mathrm{QF}(U)$ alone. Therefore we may extend the coefficient field by transcendental elements $z_2, \ldots, z_n$, thus proving that $\mathrm{GCD}_D(\bar{a}, \bar{b}) = 1$ for $D = \mathrm{QF}(U)(z_2, \ldots, z_n)[x_1, \ldots, x_n]$.

We now consider the map $\Psi\colon D \to E = \mathrm{QF}(U)(z_2, \ldots, z_n)[y_1, \ldots, y_n]$ defined by $\Psi(f(x_1, \ldots, x_n)) = f(y_1, y_2 + z_2 y_1, \ldots, y_n + z_n y_1)$ (cf. (**?** , Proof of

Lemma 6.2)). Since Ψ is a polynomial ring isomorphism over the coefficient field $\mathrm{QF}(U)(z_2, \dots, z_n)$ we must have

$$\mathrm{GCD}(\Psi(\bar{a}), \Psi(\bar{b})) = 1 \tag{1}$$

either in E or more general by Gauss's lemma in $\mathrm{QF}(U)(z_2, \dots, z_n, y_2, \dots, y_n)[y_1]$.

We now can consider the resultant

$$\varrho = \mathrm{res}_{y_1}(\Psi(\bar{a}), \Psi(\bar{b})) \in U[z_2, \dots, z_n, y_2, \dots, y_n].$$

From (1) it follows as a property of the resultant that $\varrho \neq 0$. The domain homomorphism $\phi_p\colon U \to U/(p)$ extends to the polynomial rings $U[z_2, \dots, z_n, x_1, \dots, x_n]$ and $U[z_2, \dots, z_n, y_1, \dots, y_n]$ and induces an isomorphism Ψ_p. Suppose that

$$\phi_p(\varrho) \neq 0. \tag{2}$$

Then $\mathrm{GCD}(\phi_p(\Psi(\bar{a})),\ \phi_p(\Psi(\bar{b}))) = 1$ in $U/(p)(z_2, \dots, z_n, y_2, \dots, y_n)[y_1]$. We now claim that there can be no common factor in $U/(p)(z_2, \dots, z_n)[y_2, \dots, y_n]$. First, not both of the leading coefficients of $\Psi(\bar{a})$ and $\Psi(\bar{b})$ with respect to y_1 can map to zero via ϕ_p, because otherwise their resultant ϱ would also map to zero. Suppose now, without loss of generality, that

$$\phi_p(\mathrm{lc}_{y_1}(\Psi(\bar{a}))) \neq 0. \tag{3}$$

The key property of the map Ψ is that $\mathrm{lc}_{y_1}(\Psi(\bar{a})) \in \mathrm{QF}(U)(z_2, \dots, z_n)$, which yields from (3) the stronger condition $\mathrm{GCD}(\phi_p(\Psi(\bar{a})), \phi_p(\Psi(\bar{b}))) = 1$ in $U/(p)(z_2, \dots, z_n)[y_1, y_2, \dots, y_n]$. Using the isomorphism Ψ_p^{-1} with $y_1 = x_1$ and $y_i = x_i - z_i x_1$ for $2 \leq i \leq n$ we immediately obtain $\mathrm{GCD}(\phi_p(\bar{a}), \phi_p(\bar{b})) = 1$ in $U/(p)(z_2, \dots, z_n)[x_1, \dots, x_n]$. However, neither $\phi_p(\bar{a})$ nor $\phi_p(\bar{b})$ contain any z_i, so they are relatively prime in $U/(p)[x_1, \dots, x_n]$.

Finally, we observe that all $p \in U$ that satisfy (2) must divide all coefficients in U of ϱ, thus their greatest common divisor which has a finite number of prime factors. ☒

If p is a good prime, that is, $\phi_p(\gamma) \neq 0$ and $\Delta = 1$, then $g_p = \mathrm{GCD}(\phi_p(a), \phi_p(b))$, will be an associate of $\phi_p(g)$, differing from $\phi_p(g)$ by the unit $\delta = \mathrm{lc}_X(g) \bmod p$ if g_p is monic. If we knew $\mathrm{lc}_X(g)$ then we could set $g_p \leftarrow \delta \cdot g_p$ and the CRA would, given enough primes, reconstruct g. Instead one computes a multiple of g by imposing $\gamma = \mathrm{GCD}(\mathrm{lc}_X(a), \mathrm{lc}_X(b)) = \alpha \cdot \mathrm{lc}_X(g)$ as a leading coefficient. The GCD g is now recovered from $h = \alpha g$ by dividing out by $\alpha = \mathrm{cont}_X(h)$ in step 14. The algorithm terminates when we have enough good primes such that their product is greater than $2\alpha||g||_\infty$.

As mentioned previously, the same trial division idea may be applied to algorithm P where we compute a GCD in $\mathbb{Z}_p[x_1, \dots, x_n]$ with $n > 1$. We think of this ring recursively as $\mathbb{Z}_p[x_n][x_1, \dots, x_{n-1}]$. Algorithm P— with trial division is provided below. Our description of the two subroutines with trial division is such that subroutine P— can be obtained from subroutine M— by replacing the Euclidean coefficient domain $\mathbb{Z}$ in subroutine M— by the Euclidean domain $\mathbb{Z}_p[x_n]$ and the primes p chosen in step 6 of subroutine M— with linear polynomials $x_n - \alpha$. Lemma 1 and Lemma 2 also hold. Again, for notational convenience we define $\phi_{x=\alpha} : \mathbb{Z}_p[x] \to \mathbb{Z}_p$ to be the evaluation ring morphism $\phi_{x=\alpha}(a) = a(\alpha)$. When applied to a polynomial in $\mathbb{Z}_p[x_n][x_1, \ldots, x_{n-1}]$ we mean it is applied to the coefficients of the polynomial in $\mathbb{Z}_p[x_n]$.

Algorithm P— with Trial Division

Input: $a, b \in \mathbb{Z}_p[x_n][X = x_1, \ldots, x_{n-1}] \setminus \{0\}$
Output: $g = \mathrm{GCD}(a, b)$, $\bar{a} = a/g$, $\bar{b} = b/g$

0 If $n = 1$ then compute $g \leftarrow \mathrm{GCD}(a, b)$ using the Euclidean algorithm and output $g, a/g, b/g$.

1 $(ca, cb) \leftarrow (\mathrm{cont}_X(a), \mathrm{cont}_X(b))$, $c \leftarrow \mathrm{GCD}_{\mathbb{Z}_p[x_n]}(ca, cb)$.

2 $(a, b) \leftarrow (a/ca, b/cb)$,

3 $\gamma \leftarrow \mathrm{GCD}_{\mathbb{Z}_p[x_n]}(\mathrm{lc}_X(a), \mathrm{lc}_X(b))$.

4 $\mathbf{e} \leftarrow \min(\deg_X a, \deg_X b)$.

5 $(m, g_m) \leftarrow (1, 0)$.

Loop 6 Choose $\alpha \in \mathbb{Z}_p$ at random such that $(x_n - \alpha) \nmid m \times \gamma$.

7 $g_\alpha \leftarrow \mathrm{GCD}(\phi_{x_n=\alpha}(a), \phi_{x_n=\alpha}(b))$ using algorithm P—.

8 If $\deg(g_\alpha) =_X \mathbf{0}$ then output c and stop.

9 If $\deg(g_\alpha) >_X \mathbf{e}$ then (skip this prime) goto Loop.

10 $g_\alpha \leftarrow \gamma(\alpha) \cdot \mathrm{lc}_X(g_p)^{-1} \cdot g_\alpha$ in $\mathbb{Z}_p[X]$.

11 If $\deg(g_\alpha) <_X \mathbf{e}$ then $(m, g_m, \mathbf{e}) \leftarrow (1, 0, \deg(g_\alpha))$.

12 $h \leftarrow \mathrm{CRA}([g_\alpha, g_m], [x_n - \alpha, m])$.

13 If $h = g_m$ then begin termination test.

14 $g \leftarrow h/\mathrm{cont}_X(h)$.

15 If $g \mid a$ and $g \mid b$ then output $c \cdot g, (ca/c) \cdot (a/g), (cb/c) \cdot (b/g)$

16 $(m, g_m) \leftarrow ((x_n - \alpha) \cdot m, h)$.

17 Goto Loop.

3. Computing GCDs in $\mathbb{Z}[x, y]$

3.1. Modified Brown's P Algorithm

We have observed two main inefficiencies in algorithm P when modified to use trial division. The first is the incremental Chinese remaindering that is done in step 12 which forces the explicit construction of many intermediate polynomials which implies a high number of storage allocations relative to the number of arithmetic operations being done in $\mathbb{Z}_p$, that is, relative to the real work. The second is the trial divisions made in algorithm P. If one is using classical division for multivariate polynomials, which is very likely the case, this will become the most expensive step in the entire GCD computation. Thus instead of doing incremental Chinese remaindering and trial division in algorithm P—, we propose the following method. In algorithm M— we first precompute (tight) upper degree bounds dg_x and dg_y for, respectively, $\deg_x g$ and $\deg_y g$. This will be done by algorithm B below by computing a univariate GCD in $\mathbb{Z}_p[x]$ and a second univariate GCD in $\mathbb{Z}_p[y]$ respectively. If x is chosen to be the main variable, we will use dg_y to determine how many evaluation points are needed (see step 12), compute all the images and then interpolate y from the images in a single step rather than incrementally. We omit entirely the trial divisions for the interpolated result arguing that the trial divisions (see statement 27) in algorithm M— are sufficient to verify that the final result is correct, however, we will need to devise an alternative strategy to ensure that unlucky primes are detected. At the same time that we compute dg_x and dg_y we also compute degree bounds on the degree of the content of g in each variable. This enables us to determine if $g = 1$ (see step 2), to reduce the number of evaluation points needed for y when the content is not 1 (see step 11) and to detect one source of unlucky primes (see step 10 and 17) and to intelligently choose the main variable in step 3 which we will discuss later. We present the new algorithm for $\mathbb{Z}[x, y]$ as a single subroutine, subroutine G.

Algorithm B

Inputs: $a, b \in \mathbb{Z}[x, y] \setminus \{0\}$.
Output: (dg, dc) satisfying $dg \geq \deg_x \mathrm{GCD}(a, b)$ and $dc \geq \deg_y GCD(\mathrm{cont}_x(a), \mathrm{cont}_x(b))$

0 Compute $\gamma \in \mathbb{Z} = \mathrm{GCD}(\mathrm{lc}_{x,y}(a), \mathrm{lc}_{x,y}(b))$

1 Choose a prime p that does not divide γ.

3 Set $a_p \in \mathbb{Z}_p[x, y] = a \bmod p$, $b_p \in \mathbb{Z}_p[x, y] = b \bmod p$.

4 Compute the contents and their GCD and make a and b primitive.
Set $ca \in \mathbb{Z}_p[y] = \mathrm{cont}_x(a_p)$, $cb \in \mathbb{Z}_p[y] = \mathrm{cont}_x(b_p)$, and $cg = \mathrm{GCD}(ca, cb)$.
Set $a_p = a_p/ca,\ b_p = b_p/cb$.

5 Compute $\gamma \in \mathbb{Z}_p[y] = \mathrm{GCD}(\mathrm{lc}_x(a_p), \mathrm{lc}_x(p_b))$.

6 Choose $\alpha \in \mathbb{Z}_p$ at random such that $\gamma(\alpha) \not\equiv 0 \bmod p$.

7 Compute $g_{p\alpha} \in \mathbb{Z}_p[x] = \mathrm{GCD}(a_p(x, y=\alpha), b_p(x, y=\alpha))$.

8 Output $\deg_x g_{p\alpha}$, $\deg_y cg$.

Algorithm G

Inputs: $a, b \in \mathbb{Z}[x, y] \setminus \{0\}$
Output: $g = \mathrm{GCD}(a, b)$, $\bar{a} = a/g$, $\bar{b} = b/g$

1 Compute integer contents and their GCD and make a and b primitive.
Set $ca, cb = \mathrm{cont}_{x,y}(a), \mathrm{cont}_{x,y}(b),\ c = \mathrm{GCD}_{\mathbb{Z}}(ca, cb),\ (a, b) = (a/ca,\ b/cb)$.

2 Compute degree bounds dg_x, dc_y satisfying $dg_x \geq \deg_x g$ and
$dc_y \geq \deg_y \mathrm{GCD}(\mathrm{cont}_x(a), \mathrm{cont}_x(b))$ using algorithm B.
If $dg_x = dc_y = 0$ then output c, $(ca/c) \cdot a$, $(cb/c) \cdot b$.

Similarly compute degree bounds dg_y, dc_x satisfying $dg_y \geq \deg_y g$ and
$dc_x \geq \deg_x \mathrm{GCD}(\mathrm{cont}_y(a), \mathrm{cont}_y(b))$ using algorithm B.
If $dg_y = dc_x = 0$ then output c, $(ca/cg) \cdot a$, $(cb/cg) \cdot b$.

3 Choose the main variable x and the minor variable y.

4 Compute the correction coefficient for $\mathbb{Z}$.
Set $\gamma = \mathrm{GCD}(\ \mathrm{lc}_{x,y}(a), \mathrm{lc}_{x,y}(b)\ )$

5 To ensure that the degree of the GCD computed mod p will not be less than the degree of g in both x and y, p must not divide the integer badprimes.
Set badprimes $= \gamma\ \mathrm{GCD}(\ \mathrm{lc}_{y,x}(a), \mathrm{lc}_{y,x}(b)\ )$

6 Set $m = 1$ the product of the moduli.

Nextprime

7 Choose a new prime p that does not divide *badprimes*.

8 Compute $a_p = a \bmod p,\ b_p = b \bmod p$.

9 Compute the contents in $\mathbb{Z}_p[y]$ and their GCD.
Set $ca_p = \mathrm{cont}_x(a),\ a_p = a_p/ca_p,\ cb_p = \mathrm{cont}_x(b),\ b_p = b_p/cb_p,\ cg_p = GCD(ca_p, cb_p)$.

10 Detect unlucky primes.
If $\deg_y cg_p > dc_y$ then goto Nextprime.
If $\deg_y cg_p < dc_y$ then (bound and all previous primes were unlucky) set $dc_y = \deg_y cg_p, m = 1$ and continue.

11 Compute the correction coefficient in $\mathbb{Z}_p[y]$.
Set $\gamma_y = \mathrm{GCD}(\mathrm{lc}_x(a_p), \mathrm{lc}_x(b_p))$.

12 Bound the number of evaluation points needed less 1.
Set $N = \min(\deg_y a - \deg_y ca_p, \deg_y b - \deg_y cb_p, dg_y - dc_y + \deg_y \gamma_y)$.

13 Set $n = 0$.

Nexteval

14 Choose a new evaluation point $\alpha_n \in \mathbb{Z}_p$ at random such that $\gamma_y(\alpha_n) \not\equiv 0 \bmod p$. If no such evaluation point exists then algorithm G fails.

15 Set $a_{pn} = a_p(x, y{=}\alpha_n) \bmod p$, $b_{pn} = b_p(x, y{=}\alpha_n) \bmod p$.

16 Compute $g_{pn} \in \mathbb{Z}_p[x]$ the monic GCD of a_{pn} and b_{pn} using the Euclidean algorithm.

17 Detect unlucky prime p
If $n = 0$ and $\deg_x(g_{pn}) > dg_x$ then goto Nextprime.

18 Detect unlucky evaluation points.
If $\deg_x(g_{pn}) > dg_x$ then goto Nexteval.
If $\deg_x g_{pn} < dg_x$ then set $m = 1$, $dg_x = \deg_x g_{pn}$ and goto Nextprime.

19 Leading coefficient correction.
Set $g_{pn} = \gamma_y(\alpha_n) \cdot g_{pn} \bmod p$.

20 Set $n = n + 1$. If $n \leq N$ then goto Nexteval.

21 Interpolate y.
$g_p = \text{Interpolate}([\alpha_0, \dots, \alpha_N], [g_{p0}, \dots, g_{pN}])$.

22 Remove spurious content in $\mathbb{Z}_p[y]$, and multiply by true content.
$g_p = cg_p \times \text{pp}_x(g_p)$.

23 Update dg_y
If $\deg_y g_p < dg_y$ then set $dg_y = \deg_y g_p$.
If $\deg_y g_p > dg_y$ then goto Nextprime.

24 Leading coefficient correction in $\mathbb{Z}$.
$g_p = \gamma \cdot \text{lc}_{x,y}(g_p)^{-1} \cdot g_p \bmod p$ (using the symmetric range for $\mathbb{Z}_p$)

25 First image?
If $m = 1$ then set $m = p$, $h = g_p$ and goto Nextprime;

26 Combine the new image with the old using the CRA.
Set $g_m = \text{CRA}([g_p, h], [p, m])$, $m = p \times m$.

27 If $g_m \neq h$ then set $h = g_m$ and goto Nextprime.

28 We very probably have enough primes.
Set $g = \text{pp}_{x,y}(g_m)$.
If $g|a$ and $g|b$ then output $c \cdot g, (ca/c) \cdot (a/g), (cb/c) \cdot (b/g)$

29 Goto Nextprime - either we do not have enough primes yet or all primes were unlucky.

Proof of Correctness

We will prove firstly that unlucky primes will eventually be detected in subroutine G, and secondly that the number of evaluation points $N + 1$ chosen in step 12 is sufficient. We will also explain why in step 5 we insist that p not divide $\text{lc}_{y,x}(g)$. We will not prove that unlucky evaluation points are detected since the logic is the same as that of algorithm P—.

To prove that unlucky primes are detected we state some additional properties for the polynomial g_p appearing in Lemma 1. We state Lemma 3 for the general case but we will apply it to the bivariate case $\mathbb{Z}[y][x]$.

LEMMA 3: *Let E be a Euclidean ring, $p \in E$ be a prime and $\phi_p : E \to E/\langle p\rangle$ be the modular ring morphism $\phi(a) = a \bmod p$. Let $a, b \in E[y][X = x_1, \ldots, x_n] \setminus \{0\}$. Let g be any greatest common divisor of a and b. Thus $a = g\bar{a}$ and $b = g\bar{b}$ for some $\bar{a}, \bar{b} \in E[y][X]$. Let $g_p = \mathrm{GCD}(\phi_p(a), \phi_p(b))$. If $\phi_p(\mathrm{lc}_{X,y}(g)) \neq 0$ then*

(i) $\deg_y(\mathrm{cont}_X(g_p)) \geq \deg_y(\mathrm{cont}_X(g))$ *and*

(ii) $\deg_X(\mathrm{pp}_X(g_p)) \geq \deg_X(\mathrm{pp}_X(g)))$.

Proof. From Lemma 1 we have (i) $g_p = \delta \cdot \phi_p(g) \times \Delta$ where $\Delta = GCD(\phi_p(\bar{a}), \phi_p(\bar{b}))$ and (ii) $\deg_{X,y}(g_p) \geq_{X,y} \deg_{X,y}(g)$. For any polynomial $a \in E[y][X] \setminus \{0\}$, we may write $a = \mathrm{cont}_X(a) \times \mathrm{pp}_X(a)$ where $\mathrm{cont}_X(a) \in E[y]$ is the GCD of the coefficients of a taken in X. Hence

$$g_p = \delta \cdot \phi_p(\mathrm{cont}_X(g) \times \mathrm{pp}_X(g)) \times \Delta = \delta \cdot \phi_p(\mathrm{cont}_X(g)) \times \phi_p(\mathrm{pp}_X(g)) \times \Delta$$

and

$$\deg_{X,y}(g_p) \geq_{X,y} \deg_{X,y}(\mathrm{cont}_X(g)) + \deg_{X,y}(\mathrm{pp}_X(g)).$$

Since $\phi_p(\mathrm{lc}_{X,y}(g)) \neq 0$ and $\mathrm{lc}_{X,y}(g) = \mathrm{lc}_{X,y}(\mathrm{cont}_X(g)) \cdot \mathrm{lc}_{X,y}(\mathrm{pp}_X(g))$ then $\deg_y(\phi_p(\mathrm{cont}_X(g))) = \deg_y(\mathrm{cont}_X(g))$ and $\deg_X(\phi_p(\mathrm{pp}_X(g))) = \deg_X(\mathrm{pp}_X(g))$. Since $\Delta \neq 0$ (i) and (ii) follow. ⊠

Step 7 in algorithm G ensures $\phi_p(\gamma) \neq 0 \implies \phi_p(\mathrm{lc}_{x,y}(g)) \neq 0$ hence Lemma 1 and Lemma 3 hold in $\mathbb{Z}[y][x]$. Suppose p is an unlucky prime. Then $\deg_{x,y} \Delta \geq_X \mathbf{0}$ by definition. Factoring $\Delta = \mathrm{cont}_x(\Delta) \times \mathrm{pp}_x(\Delta)$ we have either $\deg_y \mathrm{cont}_x(\Delta) > 0$ or $\deg_x \mathrm{pp}_x(\Delta) > 0$, or both. We consider each case separately.

CASE 1 ($\deg_y \mathrm{cont}_x(\Delta) > 0$*)* : From Lemma 3 (i) we have $\deg_y(\mathrm{cont}_x(g_p)) \geq \deg_y(\mathrm{cont}_x(g))$ hence we cannot lose any degree of the content when we compute a GCD under ϕ_p. Thus in step 2 in algorithm G, dc_y, which is computed by algorithm B, satisfies $dc_y \geq \deg_y(\mathrm{cont}(g))$ and in step 9 of algorithm G $\deg_y(cg_p) \geq \deg_y(\mathrm{cont}(g))$. Thus step 10 in algorithm G will detect that p is unlucky as soon as we have a prime q such that $\deg_y \mathrm{cont}_x \phi_q\ g_q = \deg_y \mathrm{cont}_x g$. But note that the tests in step 10 will not only rule out unlucky primes, primes such that $\deg_y \mathrm{cont}_x \Delta > 0$, but also good primes for which $\deg_y\ \mathrm{cont}_x\ \phi_p\, \mathrm{pp}_x(g) > 0$. Let $\mathrm{pp}_x(g) = \sum_{i=0}^{m} g_i x^i$ for $g_i \in \mathbb{Z}[y]$. These latter primes are characterized as those that divide the integer GCD($\mathrm{res}_y(g_i, g_j)$ for $0 \leq i < j \leq m$) for the case $m > 0$ and those that divide all coefficients of $g_0(y)$ in the case $m = 0$. Since the number of these primes is finite the algorithm terminates.

CASE 2 ($\deg_x \mathrm{pp}_x(\Delta) > 0$*)* : From Lemma 3 (ii) we have $\deg_x(\mathrm{pp}_x(g_p)) \geq \deg_x(\mathrm{pp}_x(g)) = \deg_x(g)$ Furthermore, when we evaluate at $y = \alpha$ in step 7 of algorithm B and at $y = \alpha_n$ in step 15 of algorithm G, Lemma 1 holds for we have that $\phi_{y=\alpha}(\gamma) \neq 0$ from step 6 in algorithm B and $\phi_{y=\alpha_n}(\gamma_y) \neq 0$ in step 14 of algorithm G, respectively, hence we cannot lose any degree in x when we compute the GCD under $\phi_{y=\alpha} \circ \phi_p$ in algorithm B or under $\phi_{y=\alpha_n} \circ \phi_p$ in algorithm G. Thus dg_x satisfies $dg_x \geq \deg_x(g)$ in algorithm B and in step 16 in algorithm G the univariate GCD g_{pn} satisfies $\deg_x(g_{pn}) \geq \deg_x(g)$. Step 17 and 18 will detect any unlucky primes as soon as we have a prime for which $\deg_x g_{pn} = \deg_x g$. But note that this test in step 17 will also rule out a good prime for which $y = \alpha_0$ is an unlucky evaluation point for we cannot tell at this point whether we have an unlucky prime or an unlucky evaluation point.

ALLAN

Finally, suppose p is an unlucky prime with $\deg_y(\Delta) > 0$. Since the number of evaluation points used (see step 15) in algorithm G is a function of dg_y (see step 12) we would like to reduce dg_y to be $\min(dg_y, \deg_y(g_p))$ (see step 23). To be able to do this we ensure that (see step 5) $\phi_p(\mathrm{lc}_{y,x}(g)) \neq 0 \implies \phi_p(\mathrm{lc}_y(g)) \neq 0$ so from Lemma 1 we have $\deg_y(g_p) \geq \deg_y(g)$.

3.2. Asymptotic Comparison

We determine the asymptotic complexity of four GCD algorithms for $\mathbb{Z}[x, y]$, namely, Brown's algorithm, our improved version of Brown's algorithm, the heuristic algorithm GCDHEU of Char et al. (5), and the EEZ-GCD algorithm of Wang (16) for the two classes of GCD problems, the balanced GCD problems and small GCD problems, defined in section 2, and for each class, dense and semi-sparse polynomials. The complexity results that we give assume that the cost of arithmetic in $\mathbb{Z}_p$ is O(1), that no unlucky primes or evaluation points occur in any of the algorithms, that the correction coefficients γ are not significantly larger than $\mathrm{lc}(g)$, and, because we are interested in the complexity of GCD algorithms as they are typically implemented in systems, we assume classical algorithms are used for arithmetic in $\mathbb{Z}$ and $\mathbb{Z}_p[x]$. Also, we have dropped any $\log\log$ factors from the complexity estimates.

The Balanced GCD Problem

For the dense balanced GCD problem we have already stated the complexity of computing $g \times \bar{a}$ using classical multiplication is $\mathrm{O}(n^6)$, computing a/g using classical division is also $\mathrm{O}(n^6)$, and Brown's algorithm is $\mathrm{O}(n^4)$.

Result 1: The asymptotic complexity of Brown's algorithm for computing g the GCD of a and b is $\mathrm{O}(n^4)$.
Proof: Brown's algorithm uses $\mathrm{O}(n)$ primes since $2M \leq B \leq 2M^2$ where $M = \max(||a||_\infty, ||b||_\infty)$ is $\mathrm{O}(n)$ digits in length. Each modular reduction in algorithm M is $\mathrm{O}(n^3)$ because there are $\mathrm{O}(n^2)$ coefficients to reduce and each is $\mathrm{O}(n)$ digits long. Since this is done $\mathrm{O}(n)$ times the total cost is $\mathrm{O}(n^4)$.

To compute each modular GCD in algorithm P (in $\mathbb{Z}_p[x, y]$) $\mathrm{O}(n)$ evaluation points for y are required since $n = \max(\deg_y(a), \deg_y(b)) \leq B \leq 2\max(\deg_y(a), \deg_y(b)) =$

$2n$. Each evaluation costs $O(n^2)$, each univariate GCD costs $O(n^2)$, hence this is $O(n^3)$ in total. The total Chinese remaindering cost in algorithm P is $O(n^3)$, $O(n^2)$ per coefficient in $\mathbb{Z}_p[y]$ and there are $O(n)$ coefficients in each of $g, \bar{a}, \bar{b}$ of degree n. Thus the total cost of the $O(n)$ modular GCD computations done in algorithm P is $O(n^4)$.

Since the integer coefficients in $g, \bar{a}, \bar{b}$ are of size $O(n)$ digits, the Chinese remaindering in subroutine M is $O(n^2)$ per integer coefficient. Since there are $O(n^2)$ coefficients in $g, \bar{a}, \bar{b}$ this is also $O(n^4)$.

Result 2: The asymptotic complexity of our improved version of Brown's algorithm is $O(n^4) + D(n)$ where $D(n)$ is the cost of dividing a/g and b/g.
Proof: The $O(n^4)$ term is essentially the same as Brown's algorithm with the main difference being that polynomial interpolation replaces the incremental Chinese remaindering but both interpolation and Chinese remaindering are quadratic. Note the initial two degree bound calculations done by algorithm B cost $O(n^3)$.

Result 3: The asymptotic complexity of algorithm GCDHEU is $O(n^6) + D(n)$ where $D(n)$ is the cost of dividing a/g and b/g.
Proof: In order to recover the variable y the evaluation point α must bound $2||g||_\infty$ which is $O(n)$ digits in length. The algorithm uses $\alpha = 2\min(||a||_\infty, ||b||_\infty)$ which is $O(n)$ digits in length. In order to recover the x variable the evaluation point β must bound $||g(y=\alpha, x)||_\infty$ which will be $O(n^2)$ digits in length. The algorithm uses $\beta = 2\min(||a(y=\alpha, x)||_\infty, ||b(y=\alpha, x)||_\infty)$ which is $O(n^2)$ digits in length. The integer GCD computation of $a(x=\beta, y=\alpha)$ and $b(x=\alpha, y=\beta)$ will therefore be of size $O(n^3)$ digits in length. Assuming that integer GCD is quadratic, this is $O(n^6)$.

Result 4: The asymptotic complexity of algorithm EEZ-GCD will depend on how the integer coefficients are handled and whether the algorithm uses trial division to terminate or waits until $a - g\bar{a} = 0$ over $\mathbb{Z}$. We assume here that a trial division strategy is used where trial division is only attempted if there is no change from one lifting step to the next. We assume also that the computation is done modulo p^l where p is a prime and $p^l > 2||a||_\infty$ where a is the lifting polynomial, though Chinese remaindering could be applied here to the integer coefficients. Assuming the former, the modulus p^l used will have to be $O(n)$ digits long. The asymptotic complexity of the EEZ-GCD algorithm will be no better than $O(n^6) + D(n) + F(n)$ where $D(n)$ is the cost of dividing a/g and b/g and $F(n)$ is the cost of factoring the leading coefficient of the lifting polynomial.
Proof: The most expensive step in the EEZ-GCD algorithm is the Hensel lifting. The single univariate GCD costs only $O(n^4)$ from $O(n^2)$ operations in $\mathbb{Z}_{p^l}$ each of which costs $O(n^2)$. The number of arithmetic operations done in $\mathbb{Z}_{p^l}[y][x]$ by the linear and quadratic Hensel lifting algorithms is $O(n^4)$ – see Miola and Yun (14) or Bernardin (2) – hence the result $O(n^6)$.

Suppose that the polynomials $g, \bar{a}$ and $\bar{b}$ have $O(n)$ terms (they are semi-sparse) then their total size is $O(n^2)$. We state without proof the results in Table 1.

[htb]

	Dense case	Semi-sparse case
Size of a, b	$O(n^3)$	$O(n^2)$
Multiplication	$O(n^6)$	$O(n^4)$
Division	$O(n^6)$	$O(n^4)$
Brown	$O(n^4)$	$O(n^4)$
Improved Brown	$O(n^4) + D(n)$	$O(n^4)$
GCDHEU	$O(n^6) + D(n)$	$O(n^6)$
EEZ-GCD	$O(n^6) + D(n) + F(n)$	$O(n^5) + F(n)$

Table 1: Complexity Results for Balanced GCD

As noted previously, $D(n)$, the cost of the trial divisions is $O(n^6)$ for the dense case and $O(n^4)$ for the semi-sparse case if classical integer and polynomial arithmetic is used. Although the cost of division for the balanced GCD problem can can be reduced to $O(n^4)$ for both cases if a modular algorithm is also used for division in $\mathbb{Z}[x, y]$, but note that the modular algorithm will also be $O(n^4)$ on the small GCD problem which is much worse than classical division.

Note that these asymptotic results do not capture the fact that Brown's algorithm will typically use three times as many primes in algorithm M than in algorithm M— because the bound B is three times as long as $||g||_\infty$, and similarly, three times as many evaluation points in algorithm P than in algorithm P—, and further, it will do three times as much work in lifting $g, \bar{a}, \bar{b}$ in both algorithms on the the balanced GCD problems. For $\mathbb{Z}[x, y]$ this is a factor of 81 which is clearly of practical significance.

The Small GCD Problem

We consider two cases. In the dense case $\bar{a}$ and $\bar{b}$ are dense of degree n in both variables with n digit coefficients. In the semi-sparse case $\bar{a}$ and $\bar{b}$ have $O(n)$ non-zero terms and are of degree n in both variables with n digit coefficients. In both cases the GCD g has degree $O(1)$ in both variables and coefficients of size $O(1)$ digits.

[htb]

	Dense case	Semi-sparse case
Size of a, b	$O(n^3)$	$O(n^2)$
Multiplication	$O(n^3)$	$O(n^2)$
Division	$O(n^3)$	$O(n^2)$
Brown	$O(n^4)$	$O(n^4)$
Improved Brown	$O(n^3)$	$O(n^2)$
GCDHEU	$O(n^6)$	$O(n^6)$
EEZ-GCD	$O(n^4) + F(n)$	$O(n^4) + F(n)$

Table 2: Complexity Results for Small GCD

This is the case where our improved version of Brown's algorithm is clearly the best overall and, importantly, it has the same complexity as classical polynomial division and multiplication. It will in both cases, with high probability, use O(1) primes and O(1) evaluation points. The cost is dominated by the $O(n^3)$ modular reduction cost in the dense case since there are $O(n^2)$ coefficients to reduce, each of which is $O(n)$ digits long. In the semi-sparse case the modular reduction is $O(n^2)$ since there are $O(n)$ coefficients each of which is $O(n)$ digits long. The evaluation and univariate GCD cost is also $O(n^2)$. The $O(n^6)$ behaviour of GCDHEU is because the integer GCD is still of size $O(n^3)$ digits in length. The $O(n^4)$ behaviour of algorithm EEZ-GCD in both cases is because the univariate GCD does $O(n^2)$ arithmetic operations on integers of size $O(n)$ digits in length.

The use of Assymptotically Fast Algorithms

Some of the general purpose computer algebra systems now use assymptotically fast algorithms for integer multiplication and long division. If an asymptotically fast algorithm for integer GCD were used, the asymptotic complexity of algorithm GCDHEU could be reduced to as little as $O(n^3 \log^6 n)$ for all cases if also the evaluation point is chosen to be a power of the base of the integer representation to permit linear time evaluation and single point interpolation.

Other Classes of GCD Problem

Although these results suggest that the modified version of Brown's algorithm will always be superior to the EEZ-GCD algorithm, this is not always the case. If the polynomials a and b are sparse enough, the Hensel based approach will beat the dense modular approach. Also, if the size of a is asymptotically larger than the size of b which in turn is assymptotically larger than the size of g and the size of g is not O(1), then because Hensel lifting may use b as the lifting polynomial, the work it does may be asymptotically less than the modified version of Brown's algorithm, even in the case of dense polynomials. Problems of this type do occur in practice. But there is another solution for this kind of problem, namely, to factor the smaller polyomial and use trial divsion to determine the GCD. Gaston Gonnet modified the Maple GCD implementation over $\mathbb{Z}$ in Maple V Release 4 to do this. This improvement was very effective.

3.3. Implementation

We have implemented our improved version of Brown's algorithm for $\mathbb{Z}[x, y]$ in Maple 6. In the following we will refer to it as the DIVBRO algorithm. We make an empirical comparison of it with the implementation of the GCDHEU and EEZ-GCD algorithms in Maple for both the balanced GCD problem and small GCD problems. The `divide` command in Maple 6 for polynomial division in $\mathbb{Z}[x, y]$ uses the classical algorithm – a modular algorithm is not used. We provide some details of the DIVBRO and GCDHEU implementations in Maple.

3.3.1. The DIVBRO Implementation

Data Structure for $\mathbb{Z}_p[x, y]$

The overall efficiency of the modular GCD algorithms depends critically on the data representation chosen for $\mathbb{Z}_p[x, y]$. As mentioned previously, the primes are chosen to be machine primes so that arithmetic in $\mathbb{Z}_p$ can be done directly by the hardware. But this is not sufficient to obtain an efficient implementation. If any of the three main steps, evaluation of $a(x, y)$ and $b(x, y)$ at $y = \alpha \in \mathbb{Z}_p$, the univariate $\mathrm{GCD}(a(x, \alpha), b(x, \alpha))$ computation in $\mathbb{Z}_p[x]$ and interpolation of y from the images, each of which is quadratic time in n, makes a quadratic number of storage allocations or function calls, the efficiency of the modular algorithm will be compromised. In Maple, we represent polynomials in $\mathbb{Z}_p[x, y]$ recursively as a Maple lists (vectors) of polynomials in $\mathbb{Z}_p[y]$. The representation for $\mathbb{Z}_p[y]$ is described in (15). It is a dense representation where a polynomial $a = \sum_{i=0}^{n} a_i y^i \in \mathbb{Z}_p[y]$ is represented by an array of machine integers $[a_0, a_1, \ldots, a_n]$. The evaluation operation, which maps $\mathbb{Z}_p[y][x]$ into $\mathbb{Z}_p[x]$ makes one storage allocation. The GCD operation in $\mathbb{Z}_p[x]$ is coded "in-place". It does $\mathrm{O}(1)$ storage operations. The interpolation is done coefficient by coefficient. It does $\mathrm{O}(n)$ storage operations.

Choice of Main Variable

Taking into account only the information about the degree of the input polynomials in the variables, a reasonable choice would be the following. Choose x to be the main variable if

$$\deg_x a \times \deg_x b \leq \deg_y a \times \deg_y b$$

and y otherwise, thus minimizing the expected cost of the individual univariate GCD computations. Since we have precomputed degree bounds dg_x on $\deg_x g$ and dg_y on $\deg_y g$, we can more accurately estimate the evaluation, univariate GCD, and interpolation costs of the algorithm with x chosen as the main variable to be

$$\begin{aligned} (dg_y + 1)(\deg_y a \deg_x a + \deg_y b \deg_x b), \\ (dg_y + 1)(\deg_x a - dg_x + 1)(\deg_x b - dg_x + 1), \\ \text{and } (dg_y + 1) dg_x (dg_y + 1), \end{aligned}$$

respectively. We choose x to be the main variable if the sum of those three quantities is less than the corresponding calculation with the roles of x and y interchanged, and y to be the main variable otherwise.

Trailing Coefficient Normalization

In the analysis we assumed that the normalization coefficient γ is not significantly larger than $\mathrm{lc}_X(g)$. This is not always a reasonable assumption. For example, in a square-free factorization of a polynomial $a \in \mathbb{Z}[x,y]$ one computes $g = \mathrm{GCD}(a, \partial a/\partial x)$ which forces γ to be equal to $lc_{x,y}(a)$ which may well be much greater than $\mathrm{lc}_{x,y}(g)$. We recommend that an implementation test if $\delta = \mathrm{GCD}(\mathrm{tc}_{x,y}(a), \mathrm{tc}_{x,y}(b))$ is significantly less than γ in magnitude, then normalize the trailing coefficient by δ instead of the leading coefficient by γ. Note that for this to work correctly, the primes chosen must not divide $\mathrm{tc}_{x,y}(g)$. Note $\phi_p(\delta) \neq 0$ is a sufficient condition.

3.3.2. The GCDHEU Implementation

The design and implementation of the heuristic GCD algorithm in Maple has been improved several times since it was detailed by Char et al. in (5). There, Char et al. evaluate $y = \alpha$ where α is computed to be the quantity $2\min(||a||_\infty, ||b||_\infty)$. This is followed by removing the integer content from $a(x,\alpha)$ and $b(x,\alpha)$ and computing recursively the GCD of $\mathrm{pp}(a(x,\alpha))$ and $\mathrm{pp}(b(x,\alpha))$. Next the algorithm evaluates at $x = \beta$ where β is chosen to be $2\min(||\mathrm{pp}(a(x,\alpha))||_\infty, ||\mathrm{pp}(b(x,\alpha))||_\infty)$. Since it is very likely that $\alpha \gg 2||g(x,y)||_\infty$ and $\beta \gg 2||pp(g(x,\alpha))||_\infty$, the algorithm should, and from experience, almost always, succeeds. The premise of the heuristic is that a polynomial computation in one or more variables into a single large integer GCD computation which has been implemented carefully, even though Maple does not use an asymptotically fast method.

An idea, first suggested by James Davenport, is to first try to show that a and b are relatively prime by using a much smaller evaluation point. The evaluation point must be sufficiently greater than any integer root of $g(x,y)$ to avoid the possibility of returning a proper factor of g.

Another idea is to assume that most likely one of $g, \bar{a}$ or $\bar{b}$ will have height less than $\min(\sqrt{||a||_\infty}, \sqrt{||b||_\infty})$ and to lift the GCD and the cofactors. This means that evaluation points of roughly half the previous size are used at each recursive application of the algorithm.

The version of GCDHEU in Maple V Release 5 applies Davenport's idea once. If this fails, evaluation points based on the minimum of the square root of the heights of a and b are used. Maple attempts to reconstruct both the GCD and the cofactors. If this fails, the implementation tries two slightly larger evaluation points and gives up if both of those attempts are unsuccessful. The implementation also predicts the size of the integers that will be created before evaluation. If they would be larger than 8000 decimal digits†, the implementation gives up, and control is passed on to another GCD algorithm.

Thus the implementation of GCDHEU in Maple should do relatively well on the balanced GCD problems because there it will be using a very good estimate for the height of the GCD g, but it should do relatively poorly on the small GCD problems because the estimate for the height of g will be much too large in this case.

†For the purpose of making timing comparisons, we have removed the 8000 decimal digit limitation on GCDHEU

3.3.3. Timing Results

All timings were made on an SGI machine at the CECM at Simon Fraser running Maple V Release 5.0. All times reported are CPU seconds as given by the `time` command in Maple. From the asymptotic results in the previous session, the cost of the two trial divisions in our modified version of Brown's algorithm is expected to dominate the cost of the algorithm for the dense balanced GCD problems. For this reason we have reported this time separately in the column $D(n)$ and have not included in the DIVBRO time. Thus the reader can see how the GCD algorithms compare with polynomial division.

The Balanced GCD Problem

This first set of timings is for dense polynomials. The test polynomials were created as follows.

```
> c := rand(10^n):
> cA := randpoly([x,y],coeffs=c,dense,degree=n):
> cB := randpoly([x,y],coeffs=c,dense,degree=n):
> G := randpoly([x,y],coeffs=c,dense,degree=n):
> A := expand(cA*G):  B := expand(cB*G):
```

The dense option to the `randpoly` command means all terms of total degree $\leq n$ are present. The timing results in Table 3 show that the bottleneck of the new implementation of Brown's algorithm is indeed the two trial divisions.

[htb]

n	$D(n)$	DIVBRO	GCDHEU	EEZ-GCD
5	.010	.030	.036	0.119
10	.058	.081	1.324	1.99
15	.167	.194	12.85	89.58
20	1.94	.394	74.88	1016.5
25	5.05	.726		
30	12.81	1.50		
35	40.20	2.21		
40	165.6	3.10		
45	767.5	4.67		

Table 3: Dense Balanced GCD

The second set of timings is for semi-dense polynomials where $\bar{a}, \bar{b}, g$ have $n+3$ terms instead of $O(n^2)$ terms. The test polynomials were created as follows.

```
> c := rand(10^n):
> cA := c()*x^n+c()*y^n+c()+randpoly([x,y],coeffs=c,terms=n,degree=n-1):
> cB := c()*x^n+c()*y^n+c()+randpoly([x,y],coeffs=c,terms=n,degree=n-1):
> G  := c()*x^n+c()*y^n+c()+randpoly([x,y],coeffs=c,terms=n,degree=n-1):
> A := expand(cA*G):  B := expand(cB*G):
```

The timings in Table 4 confirm that our implementation of Brown's algorithm and the division algorithm are both $O(n^4)$. We comment here that the polynomials have to be very sparse before the EEZ-GCD implementation beats DIVBRO.

[htb]

n	$D(n)$	DIVBRO	GCDHEU	EEZ-GCD
10	.008	.060	1.055	.181
20	.034	.259	69.144	1.188
30	.175	.761	704.33	21.517
40	1.16	1.780		
50	0.63	4.095		
60	1.96	7.601		
70	5.81	13.181		
80	11.9	21.730		
90	24.3	33.976		
100	41.9	52.561		

Table 4: Semi-Sparse Balanced GCD

The Small GCD Problem

We consider two cases, firstly dense cofactors. The test polynomials were created as follows.

```
> c := rand(10^(N)):
> cA := randpoly([x,y],coeffs=c,dense,degree=N):
> cB := randpoly([x,y],coeffs=c,dense,degree=N):
> c := rand(10^2):
> G := randpoly([x,y],coeffs=c,dense,degree=2);
> A := expand(cA*G):  B := expand(cB*G):
```

The GCD G has coefficients with two decimal digits thus one prime is sufficient to obtain G but two are used in the new algorithm. The timings shown in Table 5 show how badly GCDHEU performs.

In the second case the cofactors are semi-sparse.

```
> c := rand(10^n):
> cA := c()*x^n+c()*y^n+c()+randpoly([x,y],coeffs=c,terms=n,degree=n-1):
> cB := c()*x^n+c()*y^n+c()+randpoly([x,y],coeffs=c,terms=n,degree=n-1):
> c := rand(10^2):
> G := randpoly([x,y],coeffs=c,dense,degree=2);
> A := expand(cA*G):  B := expand(cB*G):
```

This is the best case for the new code in comparison with GCDHEU where the evaluation points are too large. The timings in Table 6 show again how badly GCDHEU performs but also, even though this is the best case for the EEZ-GCD algorithm, DIVBRO is still much better.

[htb]

n	$D(n)$	DIVBRO	GCDHEU	EEZ-GCD
10	.010	.016	.091	.093
20	.036	.025	2.29	.222
30	.090	.046	18.9	.707
40	.154	.071	88.8	5.34
50	.286	.266	331.8	10.0
60	2.46	.157	925.4	21.6
70	1.68	.218		40.1
80	3.35	.294		214
90	20.5	.399		238
100	35.7	.527		557

Table 5: Dense Small GCD

[htb]

n	$D(n)$	DIVBRO	GCDHEU	EEZ-GCD
10	.002	.013	.068	.068
20	.006	.016	1.88	.075
30	.012	.024	13.5	.155
40	.017	.033	77.1	.387
50	.026	.045	236.4	.747
80	.046	.078		3.36
120	.104	.147		5.50
160	.172	.254		29.7
200	.925	.481		60.9

Table 6: Semi-sparse Small GCD

Computing Contents

Some GCD algorithms begin by computing and dividing out the contents in the main variable. This means that two or more recursive GCD computations in one fewer variables over $\mathbb{Z}$ are made. For example, the Maple implementation of the EEZ-GCD algorithm does this. For some problems the cost of these content calculations is significant. In particular if $g = 1$ but the contents are not 1. The timings in Table 7 show that this content computation done by the EEZ-GCD algorithm can be very significant. But what it means is that the asymptotic complexity of the EEZ-GCD algorithm is not always good even when $g = 1$. The GCDHEU timings are good because of the small evaluation points being used to prove the polynomials are relatively prime.

```
> m := iquo(n,3);  c := rand(10^m);  G := 1;
> A := randpoly(x,coeffs=c,dense,degree=m)*
       randpoly(y,coeffs=c,dense,degree=m)*
    (c()*x^m+c()*y^m+c()+randpoly([x,y],coeffs=c,terms=m,degree=m-1));
> B := randpoly(x,coeffs=c,dense,degree=m)*
       randpoly(y,coeffs=c,dense,degree=m)*
    (c()*x^m+c()*y^m+c()+randpoly([x,y],coeffs=c,terms=m,degree=m-1));
```

[htb]

n	DIVBRO	GCDHEU	EEZ-GCD
10	.006	.003	0.023
20	.006	.012	0.054
30	.012	.041	0.159
40	.023	.086	0.282
50	.030	.164	1.060
60	.048	.167	5.573
70	.068	.584	1.110
80	.101	.469	5.363
90	.143	.726	23.546
100	.144	.972	68.631

Table 7: Content GCD

4. The Modular GCD algorithm over Number Fields

In this section we consider the computation of GCDs over a number field $\mathbb{Q}(\alpha)$ using our improved version of Brown's algorithm. Let $m \in \mathbb{Z}[z]$ be the minimal polynomial for the number field $\mathbb{Q}(\alpha)$ and let $k = \deg_z m$. The computation of univariate GCDs over number fields using the modular method has been investigated by Langemyr and McCallum (13) and Encarnacion (10). Encarnacion uses the modular mapping to map elements in $\mathbb{Z}[z]/\langle m\rangle$ to $R = \mathbb{Z}_p[z]/\langle m\rangle$ for primes $p = p_0, p_1, \ldots$, which do not divide $\mathrm{lc}_z m$, computes the modular GCDs over these finite rings and employs the Chinese remainder algorithm and rational reconstruction (see (16)) to recover the rational coefficients of the monic GCD in $\mathbb{Q}[\alpha][x]$. Because R may have zero divisors (when m factors mod p), it may happen during the GCD computation over R an inverse does not exist. If this happens, the algorithm simply skips this prime. If multiplication and division in the finite ring $\mathbb{Z}[z]/\langle m\rangle$ cost $\mathrm{O}(k^2)$, then the cost of one univariate GCD of degree n over this ring is $\mathrm{O}(n^2k^2)$. The same basic approach extends directly to multivariate polynomials. Here we consider the bivariate case. We first map $\mathbb{Z}[z]/\langle m\rangle[x,y]$ to the finite ring $R = \mathbb{Z}_p[z]/\langle m\rangle[x,y]$ then compute the bivariate GCD over R. For the dense balanced GCD problem, the cost of one GCD in this finite ring will be $\mathrm{O}(n^3k^2)$.

In many (most?) practical problems, the number field will be of low degree involving one or two square-roots or cube-roots. In such cases it is not difficult to find primes such that $m(z) \equiv \mathrm{lc}_z\, m \cdot \Pi_{j=1}^{k}(z-\beta_j) \bmod p$, where $\beta_i \neq \beta_j$, that is, $m(z)$ splits into distinct linear factors over $\mathbb{Z}_p$. For example, consider $\mathbb{Q}(\sqrt{2},\sqrt{3})$. Here $m(z) = z^4 - 10z^2 + 1$. One prime in four will result in $m(z)$ splitting into linear factors. We replace the GCD computation over $\mathbb{Z}_p[z]/\langle m\rangle$ which costs $\mathrm{O}(k^2n^3)$ with k GCD computations over $\mathbb{Z}_p$ each of which costs $\mathrm{O}(n^3)$. The k images are combined again by application of the Chinese remainder theorem at a cost of $O(k^2n^2)$. Of course this is only possible when k is small for the probability that $m(z)$ splits into linear factors is one in $k!$. The gain, in our example, would theoretically be a factor of 4. But one is also replacing low degree polynomial computations in the finite ring $\mathbb{Z}_p[z]/\langle m\rangle$ with machine integer computations in

$\mathbb{Z}_p$. This is analagous to the difference between multi-precision integer operations of k words long, each of which typically involves several storage management operations and several function calls, with hardware instructions for machine arithmetic. The following experiment shows that in Maple, in addition to an expected factor of 16 slowdown in going from $\mathbb{Z}_p$ to $R = \mathbb{Z}_p[z]/(z^4 - 10z^2 + 1)$, a further slowdown of a factor of 22 occurs. Thus the potential gain is a factor of 88! Here $t1$ is the time for a large GCD over $\mathbb{Z}_p$ and $t2$ is the time for a large GCD over R.

```
> p := 46327:
> A := Randpoly(300,x) mod p:
> B := Randpoly(300,x) mod p:
> t1 := time( Gcd(A,B) mod p ):
> rof := RootOf(x^4-10*x^2+1):
> A := Randpoly(300,x,rof) mod p:
> B := Randpoly(300,x,rof) mod p:
> t2 := time( Gcd(A,B) mod p ):
> t2/(16*t1);
                        22.20192308
```

On the other hand, if the number field has relatively high degree, this overhead drops. For a similar example with the degree 16 minimal polynomial $x^{16} - 136x^{14} + 6476x^{12} - 141912x^{10} + 1513334x^8 - 7453176x^6 + 13950764x^4 - 5596840x^2 + 46225$ we obtained only a factor of 3.3 slowdown more than the expected factor of 256. Again, this is analgous to what happens with multi-precision integer arithmetic. Therefore, for number fields $\mathbb{Q}(\alpha)$ of low degree, an implementation of Brown's algorithm should map $\mathbb{Z}[z]/\langle m\rangle$ into $\mathbb{Z}_p[z]/\langle m\rangle$ and then also into into $\mathbb{Z}_p$. We have found that the additional cost of searching for primes such that $m(z)$ has k distinct roots in $\mathbb{Z}_p$ to be negligible for $k \leq 4$.

Details of Implementation

To test efficiently if a prime p splits $m(z)$ into distinct linear factors we do the following. In the general case p splits $m(z)$ into distinct linear factors if $\mathrm{GCD}(m, m') = 1 \bmod p$ and $\deg_z \mathrm{GCD}(z^p - z, m(z)) = k$ where the latter calculation can be done efficiently using binary powering with remainder. Faster tests may be applied for square roots and cube roots. For the case $m(z) = az^2 + bz + c$, p splits $m(z)$ if and only if the Jacobi symbol $\mathrm{J}((b^2 - 4ac)/p) = 1$ where the Jacobi symbol can be computed efficiently by the Euclidean algorithm. For the case $m(z) = z^3 + a$, p splits $m(z)$ if and only if (i) $\sqrt{-3} \in \mathbb{Z}_p$, that is, $\mathrm{J}((-3)/p) = 1$, or equivalently, $p \equiv 1 \bmod 3$ and (ii) $a^{p-1} \equiv 1 \bmod p$. These latter calculations are no more expensive than testing integers for primality.

If p splits $m(z)$ into distinct linear factors then the roots β_i can be obtained using the probabilistic splitting method of Rabin (**?**).

Once the image GCDs have been computed, i.e. $g_{p_i} = \mathrm{GCD}(a(z = \beta_i, x, y), b(z = \beta_i, x, y)) \bmod p \in \mathbb{Z}_p[x, y]$ we obtain the GCD in $\mathbb{Z}_p[z]/\langle m\rangle[x, y]$ by interpolation.

It can happen that one or more of the β_i are unlucky evaluation points, that is, $\deg_X \mathrm{GCD}(\bar{a}(x,y,z), \bar{b}(x,y,z)) \bmod p =_X \mathbf{0}$ but $\deg_X \mathrm{GCD}(\bar{a}(x,y,\beta_i), \bar{b}(x,y,\beta_i)) >_X \mathbf{0}$. For example, consider $\mathrm{GCD}(\bar{a} = x+y, \bar{b} = x+y+z+3)$ where $m(z) = x^2 - 2$. If we choose $p = 7$ we find that $m(z) = (z-3)(z+3) \bmod 7$. We have $\mathrm{GCD}(\bar{a}(x,y,z), \bar{b}(x,y,z)) = 1 \bmod 7$ and $\mathrm{GCD}(\bar{a}(x,y,3), \bar{b}(x,y,3)) = 1 \bmod 7$, but $\mathrm{GCD}(\bar{a}(x,y,-3), \bar{b}(x,y,-3)) = x+y \bmod 7$. Therefore if $\deg_X$ of one image is greater than that of another, we have an unlucky β_i and the algorithm should skip this prime.

The timings in Table 8 are for the balanced GCD problem in $\mathbb{Z}[z]/\langle m\rangle[x,y]$ where $m(z) = z^4 - 10z + 1$. The first column of timings is for the heuristic method of Gonnet et al. (11) which works by choosing an integer s and computing $g(x,y,s) = \mathrm{GCD}(a(x,y,s), b(x,y,s)) \bmod q$ where the integer q (possibly composite) divides the integer $m(s)$ and q is sufficiently large to enable reconstruction of $g(x,y,z)$ from a single image $g(x,y,s)$. The GCD computation in $\mathbb{Z}_q[x,y]$ in Maple done using sparse Hensel lifting. The second column of timings is for our Maple implementation that maps the computation in two steps into $\mathbb{Z}_p$ and the third column is for our Maple implementation that computes over the finite ring $\mathbb{Z}_p[z]/\langle m\rangle$.

The input polynomials were created as follows.

```
> rof := RootOf(x^4-10*x^2+1);
> c := rand(10^n):
> cA := c()*x^n+c()*y^n+c()+randpoly([x,y,rof],coeffs=c,terms=n,degree=n-1):
> cB := c()*x^n + c()*y^n + c() +
  randpoly([x,y,rof],coeffs=c,terms=n,degree=n-1):
> G := c()*x^n + c()*y^n + c() +
  randpoly([x,y,rof],coeffs=c,terms=n,degree=n-1):
> A := evala(Expand(cA*G)): B := evala(Expand(cB*G)):
```

[htb]

n	Heuristic	DIVBRO over $\mathbb{Z}_p$	DIVBRO $\mathbb{Z}_p[z]/\langle m\rangle$
5	0.32	0.13	0.70
10	1.94	0.34	5.77
15	10.61	0.91	25.88
20	47.00	1.70	68.41
25	105.53	2.94	153.86
30	294.31	4.93	
35	713.49	8.46	
40		14.42	
45		21.96	
50		43.02	

Table 8: Semi-sparse Balanced GCD over $\mathbb{Q}(\alpha)$

The following set of timings in Table 9 are for the same number field but for the balanced GCD problem with a dense GCD and cofactors. We have included the percentage of time spent performing the divisions in parentheses. The input polynomials were created as follows.

```
> rof := RootOf(x^4-10*x^2+1);
> c := rand(10^n):
> cA := randpoly([x,y,rof],coeffs=c,dense,degree=n):
> cB := randpoly([x,y,rof],coeffs=c,dense,degree=n):
> G := randpoly([x,y,rof],coeffs=c,dense,degree=n):
> A := evala(Expand(cA*G)): B := evala(Expand(cB*G)):
```

[htb]

n	Heuristic	DIVBRO over $\mathbb{Z}_p$	DIVBRO $\mathbb{Z}_p[z]/\langle m\rangle$
5	0.84	0.26 (34%)	0.99
10	23.70	1.93 (70%)	10.26
15	279.88	9.00 (84%)	52.80
20	642.68	33.7 (89%)	235.8

Table 9: Dense Balanced GCD over $\mathbb{Q}(\alpha)$

This final set of timings in Table 10 are for the same number field but for the small GCD problem with semi-sparse cofactors. In this example the Heuristic method does poorly because it does not know that the integer coefficients in g are small.

```
> rof := RootOf(x^4-10*x^2+1);
> c := rand(10^(2*n)):
> cA := c()*x^n + c()*y^n + c() +
  randpoly([x,y,rof],coeffs=c,terms=n,degree=n-1):
> cB := c()*x^n + c()*y^n + c() +
  randpoly([x,y,rof],coeffs=c,terms=n,degree=n-1):
> c := rand(100):
> G := randpoly([x,y,rof],coeffs=c,dense,degree=2):
> A := eval(Expand(cA*G)):  B := eval(Expand(cB*G)):
```

The timing results confirm that when using Brown's algorithm, it is very worthwhile to map a GCD computation over a small number field into $\mathbb{Z}_p$ instead of $\mathbb{Z}_p[z]/\langle m\rangle$. They also show that Brown's algorithm can be much faster than the heuristic algorithm on these bivariate GCD problems if the computation is mapped into $\mathbb{Z}_p$.

5. The Modular GCD Algorithm over Finite Fields

Consider a GCD problem in $R = \mathrm{GF}(q)[x_1, ..., x_n]$ where GF(q) is a finite field with q elements. Such problems occur in coding theory for example. Algorithm P

[htb]

n	Heuristic	DIVBRO over $\mathbb{Z}_p$	DIVBRO $\mathbb{Z}_p[z]/\langle m\rangle$
10	0.38	0.12	0.34
20	1.7	0.18	0.94
30	6.2	0.30	1.50
40	18.1	0.70	2.80
50	17.1	0.65	4.26
60	48.2	0.87	6.27
70	65.1	1.00	6.83
80	68.7	1.36	8.75
90	176.2	2.06	11.46
100	–	2.15	15.26

Table 10: Semi-Sparse Small GCD over $\mathbb{Q}(\alpha)$

```
z := (A^2*B^2*Z^2+A^2*B^2+B^2*Z^2+Z^2+B^2)/Z:
a := (B^2*C^2*A^2+B^2*C^2+C^2*A^2+A^2+C^2)/A:
b := (C^2*Z*B^2+C^2*Z+Z*B^2+B^2+Z)/B:
c := (Z*A*C^2+Z*A+A*C^2+C^2+A)/C:
P := Z^2*A^3*B^2*C^2*(z^2*b^2*c^2+(a+z+1)^2*(a+c^2+1)):
P := expand(P) mod 2:
```

Figure 1: Maple code for H. Dobbertin's (7) polynomial

applies directly to the case when q is prime but this restriction is not necessary. However, if q is too small, algorithm P will run out of evaluation points in GF(q) and fail. For example, consider the polynomial P in figure 1 which in expanded form has 158 terms. It is a polynomial in GF(2)$[A, B, C, Z]$. To factor this polynomial the square-free factorization step calculates the GCD$(P, \partial P/\partial A)$. Maple V Release 5 applies the subresultant algorithm because it is unable to apply the EZ-GCD algorithm over GF(2) because neither 0 nor 1 may be used for evaluation. Because the subresultant algorithm blows up, Maple does not terminate.

What do we do if q is too small for the modular GCD algorithm to be applied? One known solution is to choose evaluation points from GF(q^k), a field extension of GF(q), where q^k is sufficiently large for interpolation to be possible. An alternative approach which we propose here is to consider $R = \mathrm{GF}(q)[x_n][x_2, ..., x_{n-1}]$ and apply the modular algorithm to the coefficient ring GF$(q)[x_n]$ which is a Euclidean domain (see also (**?** , section 2.7.2)).

Let m_i, $i = 1, 2, ...$ be a sequence of monic irreducible polynomials in GF$(q)[x_n]$. It is not difficult to see that the modular algorithm can be applied to compute a GCD in R by computing image GCDs modulo the m_i, that is over the finite fields GF$(q)[x_n]/\langle m_i\rangle$, and applying the Chinese remaindering to the resulting images. Indeed algorithm P does exactly this but with linear polynomials $x_n - \alpha$.

The advantage of this approach over computing in an extension field is now demonstrated for the case of $R = \mathrm{GF}(p)[y][x]$, i.e. bivariate polynomials. Let

$g, \bar{a}, \bar{b} \in R$ where GCD($\bar{a}, \bar{b}$)=1 and $\deg_y(\bar{a}) = \deg_y(\bar{b}) = \deg_y(g) = \deg_x(\bar{a}) = \deg_x(\bar{b}) = \deg_x(\bar{g}) = n$. Consider the problem of computing the GCD($a = \bar{a}g, b = \bar{b}g$) in R. This is the balanced GCD problem. In the following complexity estimates we assume classical algorithms are used for polynomial arithmetic in GF(p)[y], i.e., multiplication, division with remainder, and GCD cost $O(n^2)$ arithmetic operations over GF(p).

The approach of choosing evaluation points in GF(p^k), an extension field of GF(p) requires that $k > \log_p(1+n)$ for there to be sufficiently many evaluation points. The cost of the modular evaluation algorithm when run over GF(p^k) will depend on how the finite field is represented. We assume polynomials of degree k are used to represent the field elements, and that multiplication and division in GF(p^k) are quadratic in k. Under these assumptions we have

THEOREM 5.1: *(i) The modular evaluation algorithm when run over* GF(p^k) *on the balanced GCD problem performs $O(k^2n^3)$ arithmetic operations in* GF(p). *(ii) The modular algorithm applied to* GF(p)[y] *on the balanced GCD problem performs $O(kn^3)$ arithmetic operations in* GF(p).

Proof. In both of the following analyses, the cost of the trial divisions in step 17, which are performed in GF(p)[y][x], is not included. If the classical algorithm is used for trial division, the cost of the trial division step will be $O(n^4)$ arithmetic operations over GF(p).[‡] Furthermore, it is assumed that all primes are good primes and for simplicity $\gamma(y) = \mathrm{lc}_x(g)$.

Suppose the modular evaluation algorithm is used and the evaluation points are chosen from GF(p^k). The algorithm must compute GCD($a(\alpha_i, x)$, $b(\alpha_i, x)$) at $O(n)$ points $\alpha_i \in$ GF(p^k) in order to interpolate t of degree n in g. Each evaluation of $a(y, x)$ at $y = \alpha_i$ costs $O(n^2)$ arithmetic operations in GF(p^k). Each image GCD costs $O(n^2)$ arithmetic operations in GF(p^k). We must interpolate n coefficients in GF(p^k)[y] of degree n, each of which costs $O(n^2)$ arithmetic operations in GF(p^k). This is $O(n^3)$, $O(n^3)$ and $O(n^3)$ arithmetic operations in GF(p^k) for the evaluations, image GCDs, and interpolations, respectively. Thus we have $O(n^3k^2)$ arithmetic operations over GF(p).

Suppose the modular algorithm is applied to the coefficient ring GF(p)[y]. To reconstruct coefficient polynomials in GF(p)[y] of degree n, we will require monic irreducibles $m_i \in$ GF(p)[y] such that $\sum_i \deg_y(m_i) \geq n$. We first claim that there are sufficiently many irreducible polynomials such that we may choose

[‡] Alternatively and faster, namely in $O(n^3)$ arithmetic operations, the algorithms can interpolate/Chinese remainder the co-factors a and b. If the degrees of their coefficients stay properly bounded, i.e., if the maximum degree of the coefficients of each co-factor and the GCD adds up to the maximum degree of the coefficients of the corresponding input polynomial, the division is certified to be exact provided the moduli cover the degrees of the coefficients of the inputs.

$d_{\max} = \max_i(\deg_y(m_i)) = O(\log_p n)$. This claim is a direct consequence of

$$y^{p^k} - y = \prod_{\substack{f \text{ irreducible in } \mathrm{GF}(p)[y] \\ \deg(f) \text{ divides } k}} f(y). \tag{4}$$

The cost of each modular reduction requires division of $O(n)$ coefficients in GF$(p)[y]$ of degree n by $m_i(y)$ of degree $d_i = O(\log_p n)$. The total cost is $O(n \sum_i (n d_i)) = O(n^3)$ arithmetic operations over GF(p), as we also can have $\sum_i d_i = O(n)$. The cost of each image GCD is $O(n^2)$ arithmetic operations over GF(p^{d_i}). For all image GCDs this is $O(n^2 \sum_i d_i^2)$ arithmetic operations over GF(p). Since $\sum_i d_i^2 \leq d_{\max} \sum_i d_i = O(n \log_p n)$ the cost of the image GCDs is $O(n^3 k)$ for any $k > \log_p n$. There are $n+1$ coefficients to reconstruct using the CRA, each of which take $O(n^2)$ arithmetic operations over GF(p). The $O(n^3 k)$ arithmetic operations over GF(p) in the image GCD computations therefore dominate the cost. ⊠

Thus we obtain an improvement of a factor of $k = O(\log_p n)$. The reason for the improvement is that we are using y instead of a new variable to represent GF(p^k). Dan Grayson has offered us another view of this trick: evaluation in GF(p^k) yields k images of the GCD, all algebraic conjugates of the extension element. Our Chinese remaindering approach shows how to make use of these conjugates and recover the coefficients in GF(p). However, Grayson's view is of value when the input polynomials are presented by black boxes (**? ?**), as so-called extended domain black boxes can allow for evaluation in GF(p^k), but there would be no easy mechanism for modular reduction of the coefficients in a single variable.

Another way to interpret this improvement is the following. Just as one could use a single large prime integer p in algorithm M, rather than using many small primes, a single prime polynomial $m \in \mathbb{Z}_p[y]$ of degree $\leq 2n$ in algorithm P would also be sufficient. In the same way we used algorithm B in algorithm G to determine an upper bound on $\deg_y(g)$ we could determine an upper bound on $\deg_y(g)$ by a single univariate GCD in GF$(p)[y]$. This leads to a good algorithm which does $O(n^4)$ arithmetic operations over $GF(p)$ for the balanced GCD problem which is the same complexity as classical multiplication and division. Thus the Chinese remainder theorem buys us a further factor of $O(n/\log_p n)$ improvement for algorithm P. For comparison, we mention that the number of arithmetic operations in GF(p) done by the subresultant algorithm is $O(n^6)$.

In practice, we have observed a better gain than the expected factor of k on problems where $p = 2$ in Maple for the following reason. This will be true for any system which implements $GF(p^k)$ using a polynomial representation. We seek in the following table to show the penalty that one pays when when one computes over GF(p^k) instead of GF(p) using polynomials to represent GF(p^k). In Maple, polynomials in GF$(p)[y]$ are represented as arrays of machine integers and arithmetic is done in-place in compiled C code. Multiplication, division, GCD, are quadratic in k. See (15) for details of the Maple representation for GF(p)[t]. In comparison, polynomials in GF(p^k)[y] are represented as Maple lists of polynomials in GF(p)[y]. Arithmetic is not done in-place and is interpreted in Maple.

Let $g, \bar{a}, \bar{b} \in \mathrm{GF}(11^k)[x]$ where $\deg_x(\bar{a}) = \deg_x(\bar{b}) = \deg_x(g) = n$. The data in column 3 of Table 11 for $T(n,k)$ were generated by timing how long Maple V Release 5 takes to calculate GCD($a = \bar{a}g, b = \bar{b}g$) for different values of k, for suitably chosen n, where the coefficients of the polynomials in GF(11^k) were generated at random.

[htb]

k	n	$T(n,k)$	$\frac{T(n,k)}{T(n,1)}$	n	$M(n,k)$	$\frac{M(n,k)}{M(n,1)}$
2	256	.96e-4	242	24	.61e-5	15.6
4	196	.31e-4	79	21	.26e-5	6.5
8	144	.95e-5	24	18	.15e-5	3.7
16	100	.29e-5	7.4	15	.18e-5	4.5
32	64	.17e-5	4.3	12	.10e-5	2.6
64	36	.99e-6	2.5	9	.75e-6	1.9
128	16	.60e-6	1.5	6	.82e-6	2.1

Table 11: $n^2k^2T(n,k)$ is the time in seconds for a GCD over GF(11^k); $n^2k^2M(n,k)$ is the time in seconds for a GCD over GF(p), where p is a prime k words (of memory) long.

The data in column 4 for $T(n,k)/T(n,1)$ indicates the penalty or loss of efficiency in calculating over GF(11^k) for $k > 1$ compared with GF(11). The factor 242 for $k = 2$ means that there is an increase in cost of a factor of 242 beyond the expected factor of 4 from k^2. The factor 1.5 for $n = 128$ means that the penalty beyond the factor of 128^2 is now only 50%. There are two reasons for this. First, the Maple code for GCD computation in GF(p^k)[x] is interpreted (the coefficient arithmetic is compiled) and this overhead is relatively high when k is small whereas the code for GCD computation for GF(p)[x] is compiled C code. Second, that whereas the data structure for polynomials in GF(p)[x] is an array of machine integers, and the GCD computation runs "in-place" with no additional memory required, this is not the case for GF(11^k)[x] where the coefficient arithmetic requires data structures to be allocated as needed.

We have attempted in columns 6 and 7 to eliminate the overhead of the Maple interpreter from the picture by timing GCDs over GF(p) instead of GF(11^k) where p is a k word prime. Again, the coefficients were generated at random. In this case the Maple code is compiled and runs in-place. We can see from column 7 that for small k there is still a penalty of a factor of 15. Thus, if one chooses to represent GF(p^k) using polynomials, there will always be a significant penalty.

We can take advantage of this in the modular GCD algorithm by increasing the degree of the irreducibles (with a negligible penalty) but requiring fewer irreducibles. We observed in our Maple implementation (for GCDs in GF(p)[y][x]) the optimal degree for the irreducibles is arround 12.

Further Considerations

NEW REMARKS If the GCD problem has more then two variables, then the recursive calls in algorithm P, which are now computing in GF(q^k) may also run out of evaluation points if q^k is too small. If this happens, our Chinese remaindering approach can be applied again, recursively. However, this will result in a further slowdown in efficiency because of a second extension. To avoid this we choose the degree of the extension k to be large enough so that all recursive calls in algorithm P will very likely have a large enough field for evaluation and interpolation. We make

$$q^k > 10 + \max_{y \in \{x_2, \dots, x_n\}} \min(\deg_y a, \deg_y b)$$

where the number 10 is to allow for evaluation points in GF(q^k) that cannot be used and unlucky evaluation points. On Dobbertin's GCD problem our implementation makes a degree 5 extension over GF(2).

Our idea can also be used in the sparse modular GCD algorithm of Zippel in (17) when the field $GF(q)$ is too small. In this case, not only do we need enough points for interpolation but we need a larger field to avoid the unlucky zero problem of Zippel's algorithm. For example, in a prototype implementation of Jennifer de Kleine and the second author, we insist that $q^k > 10^6$ which means we make a degree 20 extension over GF(2) on Dobbertin's GCD problem. Once we have computed the GCD modulo the first prime in $GF(q)[x_n]/\langle m_1(x_n)\rangle$, we compute the remaining GCDs using sparse interpolation.

6. Conclusion

Acknowledgement

We gratefully acknowledge the contribution of George Labahn. The DIVBRO implementation for $\mathbb{Z}[x, y]$ in this paper is replacing an older implementation of Brown's algorithm in Maple by Dr. Labahn which we have studied. That implementation does not minimize the number of primes and evaluation points needed.

References

[1] Bernardin, L. and Monagan, M. B., Efficient multivariate factorization over finite fields, Proceedings of AAECC-12, SLNCS, **1255** pp. 15–28, 1997.

[2] Bernardin L., On Bivariate Hensel Lifting and its Parallelization, *Proceedings of ISSAC '98*, ACM Press (1998), pp. 96–100.

[3] Buchberger B., Collins G.E., Loos R. *Computer Algebra: Symbolic and Algebraic Computation* Springer-Verlag, 1982.

[4] W. S. Brown, On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors, *J. ACM* **18** (1971), pp. 476–504.

[5] Char, B.W., Geddes K.O., Gonnet G.H. GCDHEU: Heuristic polynomial GCD algorithm based on integer GCD computation. *J. Symbolic Comp.* **7**(1), pp. 31–48, 1989.

[6] E. S. Cheb-Terrab and A. D. Roche, Abel Equations: Equivalence and New Integrable Cases *to appear in Comp. Phys. Communications*, June/2000.

[7] Dobbertin, H. Almost Perfect Nonlinear Power Functions on GF(2^n): The Niho Case. Manuscript, German Information Security Agency, 1998.

[8] C. F. Gauss, *Disquisitiones Arithmeticae*, G. Fleischer, Jr., 1801.

[9] Gruntz, D. and Monagan, M.B. Introduction to Gauss, Maple Technical Newsleter, 9, pp. 23–35, Birkhauser, 1993.

[10] M. J. Encarnacion, Computing GCDs of Polynomials over Algebraic Number Fields, *J. Symbolic Computation* **20** (1995), pp. 299–313.

[11] Geddes K.O., Gonnet G.H., Smedley T.J. Heuristic Methods for Operations with Algebraic Numbers. *Proceedings of ISSAC '88*, Springer-Verlag LNCS, **358**, 475–480, 1988.

[12] Kaltofen E., Monagan M. B., On the Genericity of the Modular Polynomial GCD Algorithm. *Proceedings of ISSAC '99*, ACM Press, pp. 59–66, 1999.

[13] L. Langemyr, S. McCallum, The Computation of Polynomial GCD's over an Algebraic Number Field, *J. Symbolic Computation* **8** (1989), pp. 429–448.

[14] Miola A., Yun D.D.Y., Computational Aspects of Hensel-type Univariate Polynomial Greatest Common Divisor Algorithms, *Proceedings of EUROSAM '74* (1974) pp. 46–54.

[15] M. B. Monagan, In-place arithmetic for polynomials over $\mathbf{Z}_n$, *Proceedings of DISCO '92*, Springer-Verlag LNCS **721** (1993), pp. 22–34.

[16] Wang P.S., The EEZ-GCD Algorithm. *ACM SIGSAM Bulletin* **14** pp. 50–60, 1980.

[17] R. Zippel, Probabilistic Algorithms for Sparse Polynomials, *Proc. EUROSAM '79*, Springer-Verlag LNCS **72** (1979), pp. 216–226.